AD-A107 533 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/G 9/2
THE IMPACT OF MULTIPROCESSOR TECHNOLOGY ON HIGH-LEVEL LANGUAGE --ETC(U)
SEP 79 A EVANS, C R MORGAN, E S ROBERTS DCA100-78-C-0028
UNCLASSIFIED BBN-4188 NL

1 OF 2
AD
A107533

Bolt Beranek and Newman Inc.

bbn

AD A107533

13 YU

Report No. 4188

Final Report: The Impact of Multiprocessor Technology on High-Level Language Design

September 1979

Prepared for:
Defense Communications Engineering Center

m

Report No. 4188

FINAL REPORT:

THE IMPACT OF MULTIPROCESSOR TECHNOLOGY ON HIGH-LEVEL LANGUAGE DESIGN

Bolt Beranek and Newman Inc.:
Arthur Evans Jr.
C. Robert Morgan
Eric S. Roberts

Harvard University:
Edmund M. Clarke

X

A

September 10, 1979

Submitted to:

Mr. Paul Cohen
Defense Communications Engineering Center
1860 Wiehle Avenue
Reston, Virginia 22090

Re: Contract No. DCA100-78-C-0028

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 BBN [redacted]-4188	2. GOVT ACCESSION NO. AD-A107533	3. RECIPIENT'S CATALOG NUMBER 9
4. TITLE (and Subtitle) 6 THE IMPACT OF MULTIPROCESSOR TECHNOLOGY ON HIGH-LEVEL LANGUAGE DESIGN,		5. TYPE OF REPORT & PERIOD COVERED Final Report, May [redacted]78 - Sep[redacted]79
		6. PERFORMING ORG. REPORT NUMBER BBN Report No. 4188
7. AUTHOR(s) 10 Arthur/Evans, Jr.; C. Robert/Morgan; Eric S./Roberts; Edmund M./Clarke	15	8. CONTRACT OR GRANT NUMBER(s) DCA100-78-C-0028
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street, Cambridge, MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE 33126K TCCP 1007 TASK 1053 0500
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Communications Engineering Cntr. 1860 Wiehle Avenue Code R810 Reston, VA 22090	11	12. REPORT DATE 10 Sep[redacted]79
		13. NUMBER OF PAGES 140
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) N/A	12 147	15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

N/A

18. SUPPLEMENTARY NOTES

DCA Contract Officer's Representative -- Paul M. Cohen

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

HOL, high-order language, language design, Ada, multiprocessors, parallel processing, Steelman

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

As the cost of processor hardware declines, multiprocessor architectures become increasingly cost-effective and represent an important and attractive area for future research. In order to exploit the full potential of multiprocessors, however, it is necessary to understand how to design software which can make effective use of the available parallelism. This report examines the software environments of several existing multiprocessor systems and assesses the impact of multiprocessor architectures on the design of high-level languages and related software methodology. In partcont.

icular, this report concentrates on an evaluation of the programming language Ada as a potential high-order language for real-time multiprocessor systems. We conclude that Ada does not, as currently designed, meet the needs for real-time embedded systems.

THE IMPACT OF MULTIPROCESSOR TECHNOLOGY ON HIGH-LEVEL LANGUAGE DESIGN

Abstract

As the cost of processor hardware declines, multiprocessor architectures become increasingly cost-effective and represent an important and attractive area for future research. In order to exploit the full potential of multiprocessors, however, it is necessary to understand how to design software which can make effective use of the available parallelism. This report examines the software environments of several existing multiprocessor systems and assesses the impact of multiprocessor architecture on the the design of high-level languages and related software methodology. In particular, this report concentrates on an evaluation of the programming language Ada as a potential high-order language for real-time multiprocessor systems. We conclude that Ada does not, as currently designed, meet the needs for real-time embedded systems.

The Impact of Multiprocessor Technology
On High-Level Language Design

Table of Contents

The Impact of Multiprocessor Technology
On High-Level Language Design

Table of Contents
(continued)

The Impact of Multiprocessor Technology
On High-Level Language Design

Table of Figures

THE IMPACT OF MULTIPROCESSOR TECHNOLOGY ON HIGH-LEVEL LANGUAGE DESIGN

1. INTRODUCTION

The possibility of using multiprocessor architecture as the basis for a powerful computing system is an attractive one for several reasons. First, a multiprocessor system can achieve significantly increased computational speed by allowing parallelism in its task structure. Second, multiprocessor architecture also offers the potential for achieving system reliability through the redundancy of its processing elements. Third, as the cost of processing components (particularly the LSI-based microprocessor) declines, the cost of adding processors to a system becomes less significant in relation to the overall system cost. In light of these advantages, interest in multiprocessor architectures has grown significantly over the past decade, and it is clear that the multiprocessor option is likely to become increasingly cost-effective and important in years to come. It is also clear that the use of multiprocessor technology has an effect on software methodology which must be considered in the design of any programming language system, such as Ada, which is intended for use in a real-time multiprocessor-based environment.

The Defense Communications Agency of the Department of Defense can reasonably expect during the next few years a continually expanding use of multiprocessors in the communications networks which it constructs and maintains. For this reason, it has contracted with Bolt Beranek and Newman Inc. to study the impact of this new technology on the design of high order programming languages. This document is the final report of the resulting study.

Section 2 of this report presents a brief survey of several multiprocessor systems which are representative of the architectures in use today and those that are likely to exist within the next three to five years. Using these systems as models, the remainder of the report examines how the structure of a high-level language is influenced by the architecture of the target system and by the nature of the applications which are appropriate to that architecture.

Section 3 outlines classical approaches to process control and the specification of concurrency which have been used or proposed for use in high-level languages. This section serves to define a variety of mechanisms which will be used to evaluate specific linguistic features and to demonstrate the range of possible structures for parallel control. Section 4 examines the constraints imposed on language features by the nature of multiprocessor systems and reevaluates the various proposed structures in light of these restrictions.

Sections 5 and 6 use these results to evaluate the parallel control facilities provided by the language Ada and to assess the practicality of using Ada as a standard language for existing multiprocessor systems. Section 5 outlines the parallel control facility provided by Ada and describes its relationship to the more theoretical structures presented in Section 3. Section 6 is an evaluation of the Ada facility in light of our experience with multiprocessor applications. To the extent that the primitives provided by Ada are judged to be inadequate for use in the environment of a practical multiprocessor, alternative structures and extended facilities which would relieve the major problems are discussed. These are presented as general conclusions in Section 7.

2. OVERVIEW OF EXISTING MULTIPROCESSOR ARCHITECTURES

In order to make it possible to use concrete examples of the influence of multiprocessor architectures on the design of high-level languages, we feel that it is important to outline the basic structure of several representative multiprocessor systems to provide a basis for comparison and generalization. To this end, we have studied the following systems:

-	Tandem/16	[Bartlett77, Katzman77]
-	Plessey 250	[Williams72]
-	BBN Pluribus	[Heart73, Ornstein75]
-	CMU C.mmp	[Wulf72]
-	CMU Cm*	[Swan77a]

These were chosen as typical of what is available and expected to be available in the short term future. Of the systems above, we feel that the BBN Pluribus and the Carnegie-Mellon systems are significantly more general in their design than the other two and we have therefore chosen to concentrate on these systems in our discussion. Of the two systems designed at Carnegie-Mellon, we have chosen to concentrate on the Cm* system, largely because it is more recent and incorporates much of the experience from the C.mmp system in its design.

The following subsections include a brief summary of the structure of each machine. We emphasize several hardware features which, in our experience, interact most strongly with the design of programming languages:

- the addressing structure
- interprocessor communication and synchronization
- I/O completion or interrupt handling

2.1 Classes of Multiprocessor Architectures

Every system studied has multiple independent identical computing units or processors, each of which is connected to memory units and I/O devices. It is the nature of this interconnection which most characterizes the differences between systems. In one configuration (the Tandem/16), each CPU has its own associated memory and I/O connections and is perhaps more accurately referred to as a multi-computer than as a multiprocessor (see Figure 1). The CPUs communicate with one another over a communications bus but cannot access remote memory. It is this omission of shared memory which, we feel, keeps the system from being a multiprocessor (although there is no precise definition of either term which is generally accepted in the field).

A second possible configuration is suggested by Figure 2. Here there is a massive switch (whose components may in fact be distributed in space) which interconnects the processors, memory units, and I/O devices. C.mmp and Pluribus are such systems, although they use radically different strategies to implement the switch. Much of the discussion in the multiprocessor literature deals with switch implementation. The matter is given little

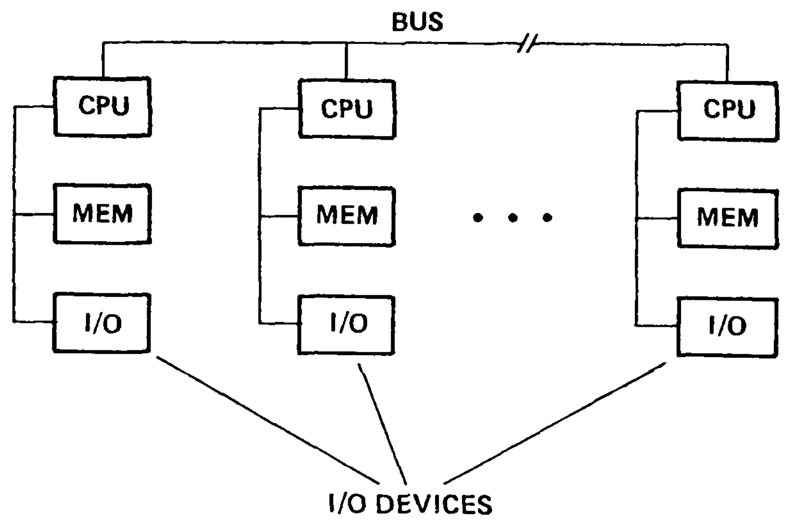

Figure 1
Typical Multi-Computer System

attention in this study, since its solution is usually invisible to the programmer.

A major problem with a fully connected arrangement as suggested by Figure 2 is that with modestly sized systems (several dozen processors) the cost and complexity of the system is dominated by the cost of the switch. An interesting variation that addresses this problem involves clusters of multiprocessors that communicate with each other, as suggested by Figure 3. Both the local switches and the master switch may be any of the customary types. Since the complexity of the master switch is based on the number of clusters rather than of CPUs, this system permits much larger configurations before the cost of the switch gets out of hand.

Often important aspects of the connection between processors and memories (at the hardware level) are invisible to the programmer, so we choose here to emphasize those features that the programmer must be aware of. For example, both Tandem/16 and Cm* hardware provide communication between processors and certain of their memories by transmitting data packets over a data communication network. In the Tandem/16, this mode of communication is visible to the programmer, who therefore perceives the system to be a distributed computation system. However, in Cm* the programmer sees shared memory. Although the hardware uses data packets for efficient communication, the

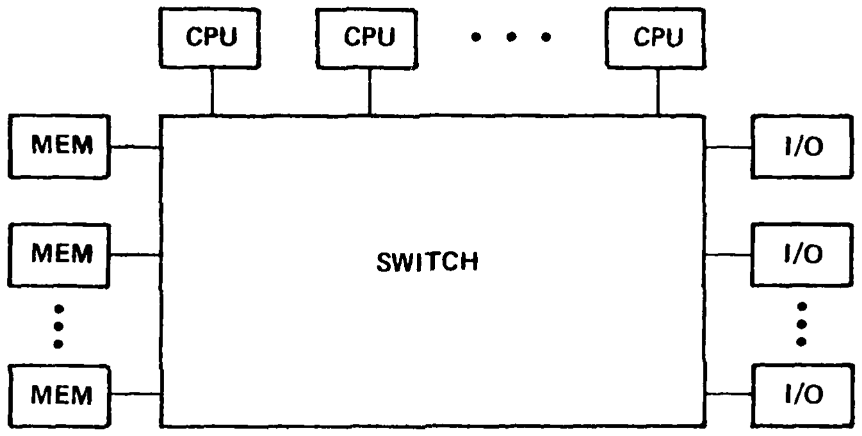

Figure 2
A Typical Fully Connected System

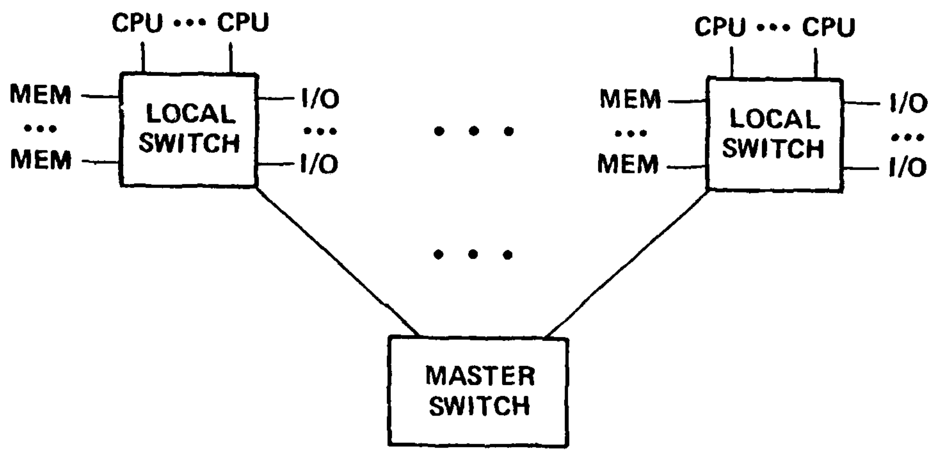

Figure 3
A Typical Cluster System

programmer sees a contiguous portion of shared memory which is homogeneously addressable. Thus the properties of the machine that are visible to the programmer are quite different from those properties of interest to hardware designers. We are interested here in only the former; as it happens, most of the literature on multiprocessors deals with the latter.

Each of these systems was designed with a particular operating system or application in mind, and the appearance of the system as viewed by its designers and as described in the literature is determined as much by this software as by the hardware. In general, we make no attempt to distinguish the hardware from the intended application, describing only the effect of the two together.

2.2 The Tandem/16 System

The Tandem/16 multiprocessor [Bartlett77, Katzman77] is designed for use as a communications processor where high reliability is required. The central component of its structure is a high speed bus to which each processor is connected. Connected to each processor are memory units and I/O devices. The only way that the processors can communicate with one another or with another processor's memory is by transmitting data packets over the bus. For reliability, each I/O device is connected to two different processors. The controlling processor is determined by an internal switch. Thus the Tandem/16 is essentially a distributed computation system similar to a set of hosts on a packet-switching network.

The only communication between processors is by packet communication and there is no shared memory. Each processor determines its functions and acts independently of all other processors. The operating system contains software to determine the failure of one or more processors and eliminate them from the configuration, leaving their functions to be performed by other processors.

2.3 The Plessey 250 System

The Plessey 250 [Williams72] is a multiprocessor designed for use as a computerized switching unit in the British telephone system. The hardware consists of multiple CPUs, memory units, and I/O devices. Each CPU resides on a separate communication bus which has an interface to each memory unit and I/O device. Thus each CPU can communicate directly with each memory and I/O device.

Besides being a multiprocessor, the Plessey 250 implements the capability concept, in the sense that a subroutine or a section of memory or a data base can be specified to be accessible only by a process which has the correct attribute or capability. For example, the programmer may specify restrictions such as the following:

(1) A data base is read only, either to everyone or to everyone except one or two processes.

(2) A subroutine is callable only by a set of privileged processes.

(3) A called routine may not access the data bases of its caller.

The Plessey 250 incorporates capabilities as a reliability feature, since the telephone system places heavy reliability constraints on switching systems. The design criteria permit a ten second down time once a week and ten minute down times once every fifty years. The capability feature is intended to prevent faulty software or hardware from drastically disabling a data structure, I/O device, or subroutine.

The Plessey 250 is designed to minimize the cost of context switching between processes. The only process switching desired is for elapsed time completed or blocking for I/O. I/O interrupts are not used. Instead, each I/O controller contains a processor which performs the task of an I/O service routine. The controller is provided a queue of messages to be sent or addresses for messages received. It then performs these functions in turn. For a device such as a disk, where complex interactions are needed between processor and controller (for example disk latency scheduling), the controller processor performs the function.

Thus the Plessey 250 can be characterized as providing shared memory for communications between processors and simple context switching. The context switch consists of storing the active registers of a process, scheduling the next process, and restoring the registers for that process from its data area.

2.4 The BBN Pluribus System

The BBN Pluribus [Heart73, Ornstein75] is a multiprocessor designed for reliable operation in environments where occasional short outages (on the order of a few seconds) are permissible, providing that the system recovers automatically and quickly. It consists of one or more processors, several memory components, and a collection of I/O devices, all of which are collected on a set of communications busses called INFIBUSes. The processors are connected to processor busses, the memories to memory busses, and the I/O devices to I/O busses. The processors, which are Lockheed SUE processors, are usually connected in pairs along with some local private memory to each processor bus. Each memory unit is connected to one memory bus, and there may be one or more memory busses. Each I/O device may be connected to either one or two I/O busses, the latter for increased reliability in case of I/O bus failure. A typical configuration for a relatively large application is shown in Figure 4.

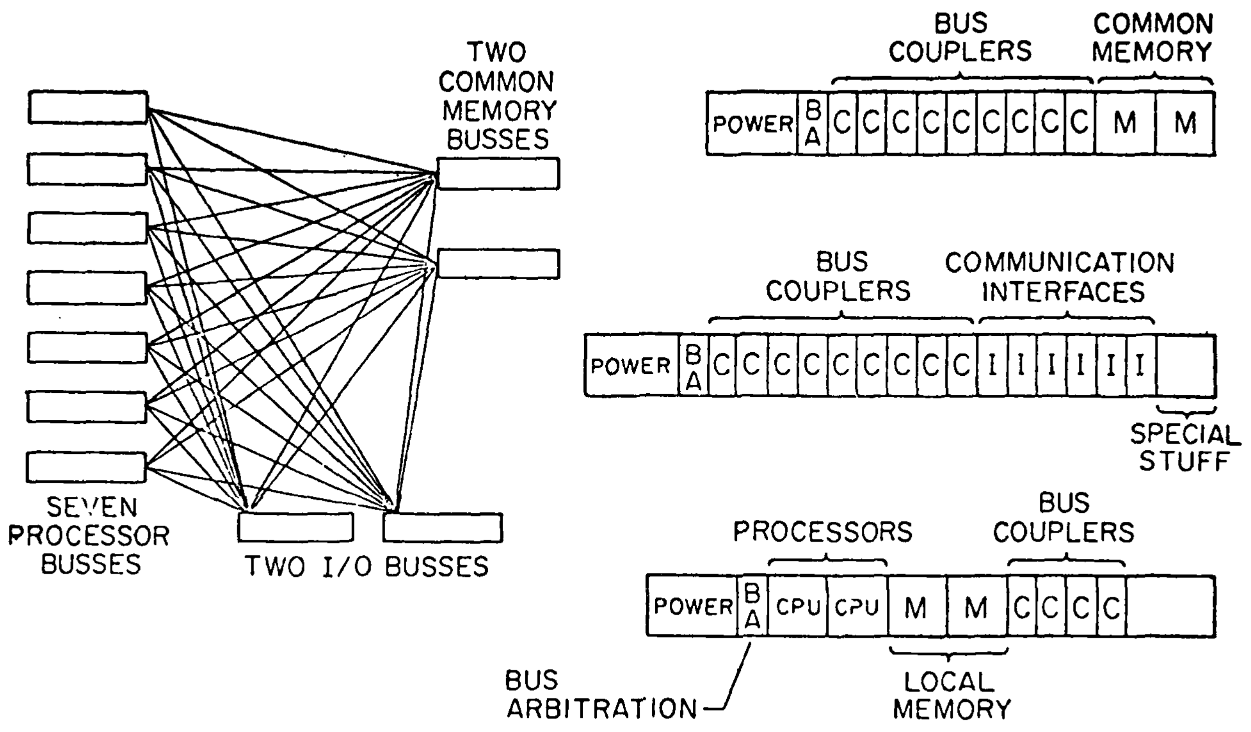

Figure 4
Typical Pluribus Configuration

The hardware organization of the Pluribus makes it possible to achieve system reliability through the redundancy of hardware components and interconnection paths. For example, if a processor bus becomes unusable through hardware failure, the remaining processors will generally be able to provide sufficient computational power to run the system.

Processor busses and I/O busses are connected to memory busses by bus couplers, each of which interprets address references in a specific range of address space and maps such addresses to memory addresses of a memory unit on the memory bus (as described in Section 2.4.1). The memory mapping is controlled by processor-controllable map registers in each bus coupler. Thus each processor can access any location in any memory unit on a processor bus. Each processor is connected to each I/O bus by a bus coupler, allowing the processors to control the I/O devices.

Pluribus was designed to provide a reliable and modularly expandable machine environment for applications such as the IMP nodes of the ARPANET. All processors in a Pluribus are identical. Each processor schedules its own activity and can execute any function performable by any other processor in the machine. Hence the Pluribus can continue to function even when one or more processors have failed. Each process within the Pluribus operating program is written to cooperate with every other process. Each process performs its own scheduling, remaining cognizant of the amount of time used since the beginning of its execution and interrupting its own operation when its time slice is completed.

2.4.1 Addressing on the Pluribus

The Lockheed SUE (which is used as the processing node for the Pluribus) is a 16-bit minicomputer and is internally capable of handling addresses which are, correspondingly, 16 bits long, restricting the processor address space to 64K bytes. This value is, unfortunately, far too small to reference directly all of the memory required for a typical application. Furthermore, the processor must be able to reference an address space which is not homogeneous; processors have local memory on the processor bus as well as access paths to common memory. These considerations force the adoption of some memory address transformation scheme which can map a small (16-bit) logical address space into a large (20-bit), non-homogeneous physical one.

In the Pluribus, the processor address space is divided into eight regions, each of which is 4K words in size. The first two regions refer to the processor's own local memory which is limited to 8K words. The final two regions indicate addresses in I/O space, including both I/O addresses which are common throughout the system and those which are local to the processor.

The remaining four regions (or windows) are used to reference common memory. Addresses 4000 through 5FFF (in hexadecimal) fall within the M0 window, 6000 through 7FFF are in the M1 window, and so on for windows M2 and M3. Each of the windows is associated with a mapping register (in the processor I/O space) which indicates the physical base address for that window in the 20-bit physical memory space.

Although this mapping scheme makes it possible to reference a 20-bit memory space using a 16-bit machine, it places some restrictions on system design. In particular, a process may not reference data on more than four pages without explicitly changing the setting of the map registers. Furthermore, since the M0 window is conventionally used to reference the user program and M3 is handled specially to allow for mutual exclusion (see below), there are only two windows available through which data may be referenced. This limitation requires that the programmer either (1) code in such a way that only two structures need to be referenced for most of a body of code, (2) maintain careful control over the locality of memory allocation to insure that all desired structures reside on the same physical page, or (3) pay the overhead of frequent changes in the map register settings.

2.4.2 Process Synchronization and Locking

In any multiprocessor system, there must be a mechanism to provide for restricted access to a data structure throughout some critical region in the code. At a primitive level, this mechanism generally depends on the existence of an indivisible read/modify/write instruction which allows a processor to interrogate the existing state of a cell in memory and change that state without interference from other processors which might also be accessing this cell.

In the Pluribus, this effect is achieved by specializing one of the mapping windows rather than through the use of a special instruction. If a memory fetch is performed using the M3 window (described above), the hardware performs a read-and-clear operation instead of a simple read. This feature is used to implement a simple locking discipline which is used as the mutual exclusion and synchronization mechanism throughout any Pluribus application. Data structures which require protection against concurrent access are governed by a lock word in common memory which is ordinarily non-zero. Before accessing data in the structure, a process reads the lock word using the M3 window. If the value read is non-zero, the process continues with exclusive rights to the structure; if the value is zero, the process continues to read the lock word (busy waits) until a non-zero value appears. At the end of the critical region, the process writes a non-zero value to the lock word which allows another process to proceed.

Although more sophisticated structures for task synchronization have been considered for the Pluribus, there is not currently any standard mechanism which provides a more powerful tool for synchronization, such as semaphores or message handling.

2.4.3 Response to External Events

One of the major questions which arises in the design of any multiprocessor is that of how to allocate the current tasks to the available processors. In the absence of any real-time constraints, this problem is relatively straightforward and can be managed by techniques similar to those used in multiprocessing systems running on a single-processor machine. If certain tasks require attention within some fairly short interval of time, some additional mechanism beyond the traditional scheduling structure is required to insure that some task (presumably with less urgent requirements) can be suspended so that its processor becomes available for the higher priority task.

In conventional systems, this problem is handled through the use of an interrupt system which generally has an internal priority mechanism so that interrupt-generating events can be responded to in their order of importance. In the Pluribus, a conscious decision was made to avoid the use of interrupts for this purpose. This decision was based on the following, substantially correct, assumptions about interrupts:

(1) Interrupt handlers tend to be incredibly difficult to write and still more difficult to debug.

(2) In a multiprocessor, it is difficult to choose which processor should field a given interrupt, particularly if reliability is an issue.

(3) There is generally no mechanism for a running task to identify points at which interrupts would be easier or harder to service other than through the relatively drastic expedient of disabling all interrupts in a region of particularly critical code.

In an attempt to avoid these problems, the designers of the Pluribus tried to use a markedly different approach. Rather than demanding immediate service through an interrupt structure, high-priority events record their need for service by making an entry in a priority queue structure which operates as the system's task manager. For efficiency, this queue structure is actually handled by a special hardware device called a PID, or pseudo-interrupt device. Each processor is required, by Pluribus convention, to check the PID every few milliseconds and service the highest priority task.

The use of a scheduling mechanism of this type leads to a distinctly different style of software organization. In the Pluribus, software is expected to be written as a collection of logical units which are referred to as strips. Each strip is associated with an entry in the PID which signifies the service priority for that task. The basic scheduling operation of the Pluribus system is managed by a simple dispatch loop which performs the following steps:

(1) Read (and erase) the highest-priority task entry in the PID device.

(2) Dispatch to the starting address for the strip corresponding to that PID entry. If the PID contained no task entries, this dispatch address simply returns to step (1) to wait for a task to appear.

(3) Whenever a strip completes its operation, it simply jumps back to step (1) to schedule a new task.

One of the most important aspects of the PID device is that task entries can be set by hardware devices as well as by software operations. Typically, when a device requires service, the device sets the entry in the PID which corresponds to its service routine. Since the PID is used for hardware devices as well as software scheduling, system responsiveness becomes an extremely important concern. Most devices can wait for only a relatively short period (on the order of a few milliseconds) before losing data; this constraint makes it essential to design the system software so that the response time requirements can be met.

The effect of the time constraint implied by the devices is to limit the length (in terms of execution time) of the software strips. The general rule for strips in the Pluribus is that no strip is allowed to run for a time which is longer than the service time requirements of the most time-critical device. This strategy guarantees that some processor will be free to perform the necessary service within the required period of time.

During the initial design phase of the Pluribus, it was assumed that the effect of the "short strip" discipline on the difficulty of software design would be relatively small in comparison to the benefits that one could achieve by avoiding interrupts and their attendant difficulties. The early experience with the Pluribus IMP seemed to bear out this assumption:

> Experience with a number of Pluribus system applications has indicated that the processor overhead and programmer effort associated with breaking tasks into strips is not a serious problem and is a relatively small price to pay for the increased reliability and performance of the novel Pluribus architecture.
>
> -- Pluribus Document 2
> page 28 [BBN75]

The success of the short strip mechanism, however, depends to a large extent on the ease of dividing logically continuous processes into strips of the appropriate size and determining where "good" spots for strip breaks occur. Normally, a programmer designs software using algorithms which may be divided more easily at some points than others.

In general, the division of a process into strips is accomplished by (1) saving any current context in global memory, (2) returning to the central dispatch routine, and (3) restoring the local state (including the control context) when the process is restarted. If the component processes are extremely simple, the amount of state information may be very small; in this case, the overhead associated with a strip division is usually quite small. For more complex processes, such as those which use a stack to manage the dynamic flow of control, the problem of saving state is considerably more expensive.

Unfortunately, the nature of the lock discipline used to manage critical resources makes the process of strip division considerably more difficult. If strips are allowed to dismiss with locked resources, deadlock situations are likely to occur. Conversely, requiring all strips to unlock all locks prior to a strip break implies that the maximum strip time is also the maximum time that may be devoted to any critical region of code. Given this constraint, it is not always possible to divide a process simply by inserting code to save the local context, dismiss the processor and restore the saved state when the process is restarted. For some applications, this forces extensive redesign of the algorithms to restrict the size of locked regions or allow concurrent access to structures.

It is easy to provide examples of tasks for which the lock discipline and strip time limitation force a departure from the natural algorithmic approach. In a dynamic memory allocation system, buffer blocks are typically allocated dynamically and circulate between various queues and the free list during system operation. A background reliability mechanism or a garbage-collection routine may try to verify the integrity of each of these structures from time to time. If the free list is long, it will not be possible to check the entire structure in one strip time and the reliability mechanism is forced to dismiss in the middle of its operation.* Clearly, if the remainder of the free list is locked at this time, allocation routines will not be able to proceed until the reliability process is rescheduled. On the other hand, if the structure is not locked, there is no way to resume the process since any preserved state is likely to have been changed in the interim. Solving the problem through cooperating schemes similar to existing strategies for concurrent garbage collection is possible, but this implies a much less natural (and considerably more difficult) algorithmic structure.

----

* Note that it is not necessary, in the single long strip approach, to lock the entire free list for the duration of the strip. If interlocks exist in each buffer, the reliability routine may simply lock chain pointers as it moves down the free list, thereby allowing the allocation routines to use previously examined buffers. The allocation process is prevented from overtaking the checking process by the individual locks, thereby insuring the integrity of the remainder of the list until each buffer has been tested.

The short strip discipline has an additional drawback which becomes clear only when the question of program structure is introduced into the discussion, such as in the context of a high-level language system. Intuitively, it seems likeley that the use of a well-designed programming language and a good optimizing compiler could provide either an automatic approach to the problem of strip division or could add structure to the software that would allow the programmer to "think" in terms of short, logically independent processes. While this is an attractive notion, most of the classical techniques used to design "well-structured" code only tend to make the problem of strip division increasingly complex. The determination of strip boundaries with minimum context not only requires that the programmer understand the exact nature of the context of each module but also forces the programmer to write in a way which is generally inconsistent with the model of hierarchical, functionally-independent routines. Such standard mechanisms as a control stack or modular subroutine structure tend to increase the amount of context and to reduce its level of visibility as a convenience to the programmer.

This problem, and its associated impact on the overall structure of the Pluribus, can be illustrated through an example drawn from any traditional operating system environment. In order to simplify the handling of I/O devices from user code, an operating system provides system calls or subroutines which perform primitive functions such as READ and WRITE. The standard interpretation of a READ call is to (1) initiate an I/O operation, (2) suspend the execution of the user process pending completion of the READ and (3) awaken the user process when the data transfer is complete. The natural place for a strip boundary in the Pluribus occurs at the point of suspension. Unfortunately, this occurs inside the subroutine READ which, in most environments, will have no idea what the current context of its caller is and will be forced to save the entire conceivable state of the calling process. There is, in addition, no way to insure that locks are properly unlocked around the system call to prevent deadlock situations from arising.

2.5 The C.mmp System

Any discussion of multiprocessor architecture would be incomplete without some mention of the C.mmp (the CMU multi-mini-processor) developed at Carnegie Mellon University in the early 70's [Wulf72, Fuller78]. This machine consists of 16 PDP-11 computers connected through a central crosspoint switch to 16 primary memory units as shown in Figure 5.

The crosspoint switch handles single word data transfers between the processors and primary memory. Since all data paths through the crosspoint switch are the same length, any processor

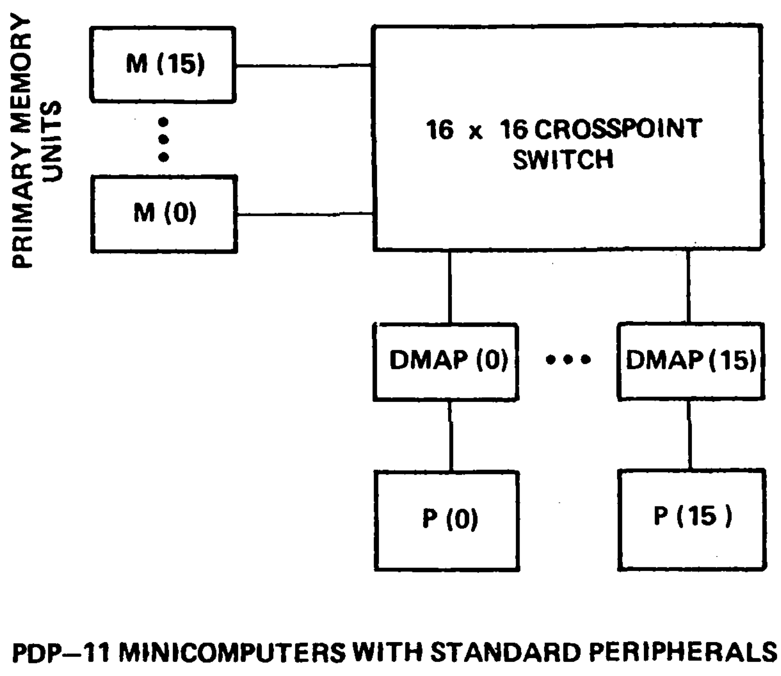
PRIMARY MEMORY UNITS
M (15)
M (0)
16 x 16 CROSSPOINT SWITCH
DMAP (0)
DMAP (15)
P (0)
P (15)
PDP—11 MINICOMPUTERS WITH STANDARD PERIPHERALS

Figure 5
Organization of C.mmp

can acess any memory unit in a constant amount of time. Parallel execution of the processors is possible because data paths through the switch are independently established for each memory reference and because multiple data paths involving different processors can exist simultaneously.

Each of the 16 memory units contains 2**16 words which gives the entire system more than a million words of primary memory. The address space of each PDP-11 processor is divided into eight 4K pages which may be physically located anywhere in the primary memory. Translation of processor-generated virtual addresses into actual primary memory references is performed by the DMAPs which connect each processor to the crosspoint switch.

2.6 The Cm* System

Cm* [Swan77a] is an experimental multiprocessor system developed at Carnegie-Mellon University to investigate processor interconnection strategies. The architecture allows large numbers of inexpensive microprocessors to be linked together so that all processors share a single virtual address space. The design also provides considerable support for low level operating system functions and interprocess communication.

The basic building block of the Cm* system is the <u>computer module</u> which consists of a Digital Equipment Corporation LSI-11 processor and a number of units of primary memory. Computer modules are organized into a hierarchical structure by a network of switches and busses so that any unit of primary memory can be accessed by any processor. This interconnection scheme is illustrated in the three figures beginning with Figure 6. At the lowest level in the hierarchy the computer modules are connected by map busses. Each module contains a switch, the Slocal, which routes memory references onto the map bus when the reference is to memory in another module (Figure 6). A maximum of 14 computer modules may be connected to a given map bus to form a cluster (Figure 7). All computer modules in a cluster share a common routing mechanism for intercluster memory references called the Kmap. The Kmaps run under microprocessor control; many operating system functions in addition to address translation are written in Kmap microcode to increase system performance.

Clusters, in turn, are connected together by intercluster busses. A configuration of four clusters is shown in Figure 8. Note that a given cluster may not be directly connected to other clusters in the same configuration. Thus, when a processor in cluster 1 references memory in cluster 4, Kmap1 must direct the reference onto the bus connecting cluster 1 and cluster 2. The reference is recognized by Kmap2 and directed onto the bus connecting cluster 2 and cluster 4. The reference is finally accepted by Kmap4 which accesses the appropriate memory location in cluster 4 and passes back an acknowledgement or data to the requesting processor in cluster 1.

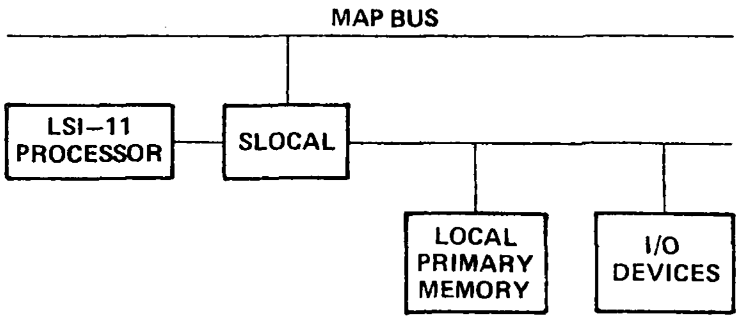

Figure 6
Structure of Cm* (Individual Modules)

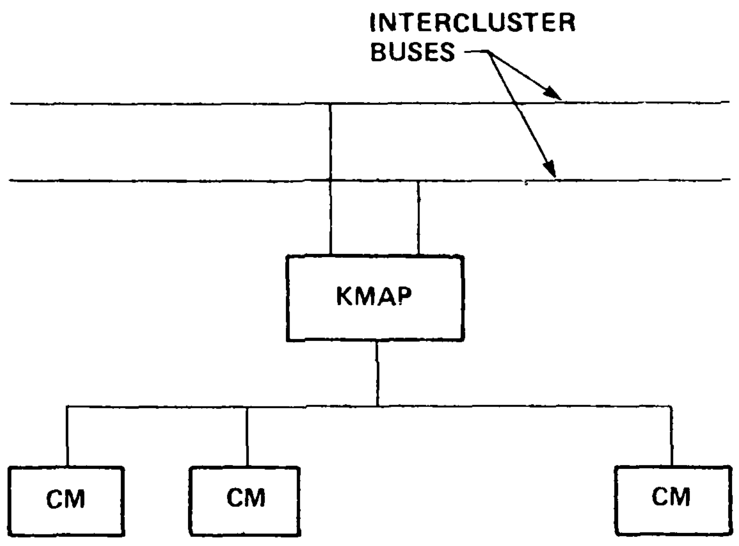

Figure 7
Structure of Cm* (Clusters)

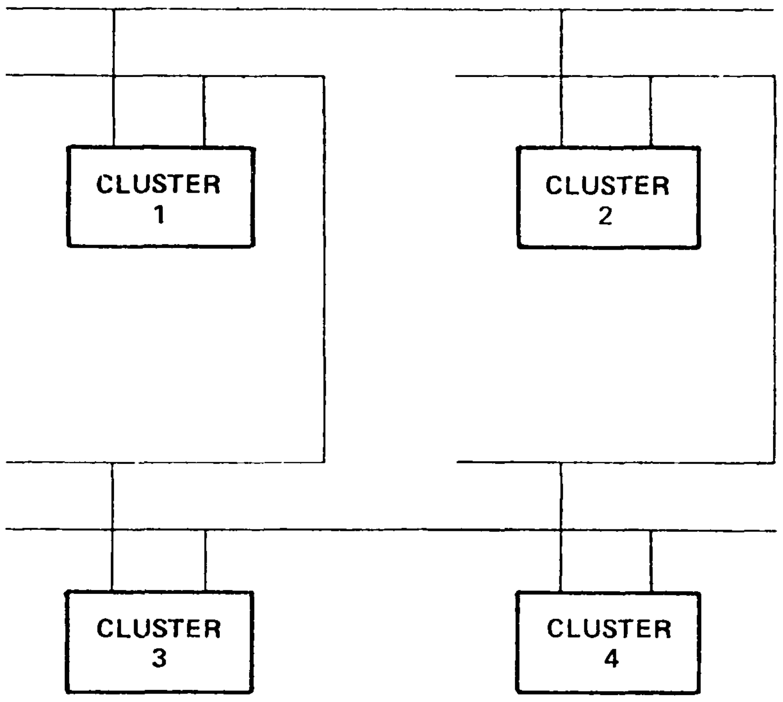

Figure 8
Structure of Cm* (Set of Four Clusters)

If a processor in cluster 1 accessed memory in cluster 4 at the same time that a processor in cluster 4 accessed memory in cluster 1, deadlock might occur over the allocation of intercluster busses. To prevent such deadlocks, packet switching is implemented at the microcode level in the Kmap processors to pass addresses and data from one cluster to another. This use of packet switching is transparent to programs running on the system. Thus, processors see a single virtual address space and can directly access memory anywhere in the system.

Under the interconnection scheme used by Cm*, there are no arbitrary limits on the size of the system since memories, processors, and Kmaps can be incrementally added as required. For the interconnection scheme to be time-efficient, however, a large fraction of each processor's memory references must be to memory which is local to the processor. Preliminary simulation studies at Carnegie-Mellon indicate that high hit ratios for local memory can be obtained if the code that a processor executes is placed in the same computer module.

2.6.1 Addressing on Cm*

The Cm* addressing scheme is strongly influenced by the fact that individual processors are LSI-11s which generate only 16-bit addresses. The 16-bit addresses are mapped onto a 2**28 byte segmented virtual address space by relocation tables and associated hardware in the Slocals and in the Kmaps. References to segments in the local memory (i.e., within the same computer module) are mapped to corresponding physical addresses directly by the Slocal. For segments which are not in the processor's local memory, the Slocal causes the processor to be delayed and a service request sent to the Kmap for the cluster in which the processor resides. If the reference is to memory within the same cluster, the processor in the Kmap generates a physical address and sends it to the appropriate Slocal. If the processor references a segment in another cluster, the processor in the Kmap will transmit the request to the desired cluster through the network of intercluster busses.

The Cm*'s virtual address space is divided into a maximum of 2**16 segments. Segments are of variable size up to a maximum of 4K bytes, and associated with each segment is a segment descriptor which specifies the physical address and length of the segment. Memory protection is based on the use of capabilities; a capability is a two word item containing the name of a segment (actually, a segment descriptor) and a rights field. Each bit in the rights field indicates whether a certain operation is permitted on the named segment. If a processor attempts to access a segment in a manner which is not permitted by the rights field in the corresponding capability, then a protection exception will be flagged. Access to capabilities is restricted so that users of the system cannot arbitrarily create or modify them.

The 64K address space of an individual LSI-11 processor is divided into 16 pages of 4K bytes each. Each page provides a window into the system virtual address space. Special registers called window registers are used to establish the binding between pages in the immediate address space of a processor and segments in the virtual address space. Each window register contains a capability for a particular segment. When a 16-bit address is generated by a processor, the top four bits are used to select a window register. Each window is associated with a capability for one of the segments in the Cm* address space. The remaining 12 bits of the address are used as an offset to specify a byte within the segment (see Figure 9). To overlay the processor's address space, i.e., to change the mapping from windows to segments, a program simply writes a new capability into the appropriate window register.

2.6.2 Process Synchronization and Communication

Cm* provides both locking and message passing primitives for synchronization and communication by user processes. Locking is used by processes for mutually exclusive access to shared data structures. In many systems, locking is provided through the use of a hardware instruction to test and set a given memory location

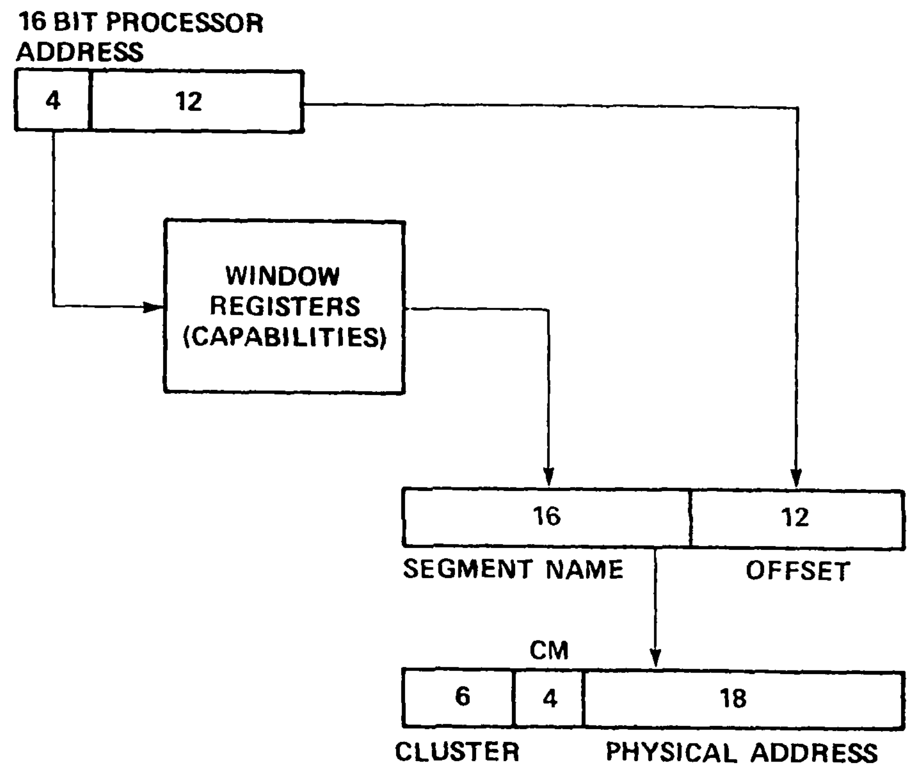

Figure 9
Addressing Structure of Cm*

without the possibility of an intervening access to the same location by another processor. On Cm* this facility is provided by the Kmap, which can lock a segment descriptor while it makes a series of references to the segment. To implement locking at the program level, two special microcoded segment operations are provided:

(1) Inspect the word addressed. If greater than zero, then decrement by one. Return the original value.

(2) Increment the word addressed by one. Return the original value.

The message passing system is also implemented by microcode running on the Kmap processors. A message is either an entire segment or a single data word encoded as a capability; <u>send</u> and <u>receive</u> operations are provided to transfer messages between processes. Although implementing message passing at the microcode level is more efficient than implementing it in software at the operating system level, send and receive operations still take considerably longer to execute than simple locking operations.

2.6.3 Response to External Events

Interrupts are handled in a more conventional manner on the Cm* than on Pluribus. Each interrupt is associated with a unique <u>interrupt vector entry</u> in main memory which indicates the appropriate response to be made when the interrupt occurs. Two options are provided for responding to interrupts. Under the first option a new process is created to execute the function named in the interrupt vector entry. This allows for the interrupt to be serviced in parallel with the continued execution of the interrupted process. Under the second option the interrupt vector entry may be used to direct status information to a specific process communication structure called a mailbox. Interrupts are then serviced sequentially by the process receiving the message.

3. LANGUAGE FORMS FOR EXPRESSING CONCURRENCY

Over the past few years, a number of experimental languages or programming constructs have been devised to support concurrent programming in the context of a time-sharing system supporting multiple processes or in the environment of a true multiprocessor system. In this section, we present a brief survey of the more significant facilities of this type.

3.1 Techniques for the Specification of Parallel Control

In addition to the mechanisms used to control the interaction of separate processes, languages designed for use in multiprocessor environments must provide a comprehensive mechanism for declaring or creating the concurrent processes. What follows is a brief overview of the more important constructs which outlines the major advantages and disadvantages of each form. On the basis of our experience in software design, it has become clear that the relativ utility of the various syntactic structures used to represent parallelism depends significantly on the nature of the parallelism expressed, so that a given syntactic form may be more appropriate in one application than it is in another.

3.1.1 Specification of Concurrent Paths

Occasionally, one is faced with an application task whose structure may be represented with a diagram such as that shown in Figure 10. In the structure shown, control operates serially as a single process until point "A" is reached. At that point, the required processing splits into several independent segments, which may be performed in parallel. When each of the parallel processes is complete, the three paths rejoin at "B" and continue with serial processing. If the parallel processes reach point "B" at different times, computation on some of the earlier paths will be suspended until all processes are ready to proceed.

An example of this type of program structure arises in the computation of arithmetic statements in which individual components in an expression may be computed independently. For example, in the expression

```
RESULT := F1(X) + F2(Y) + F3(Z)
```

the calls to the functions F1, F2 and F3 may be executed in parallel if the compiler can guarantee that the functions F1, F2 and F3 are side-effect free. This calculation gives rise to the computation graph (see Figure 11) which has the same form as the concurrent subtask structure.

```
common execution
       |
       |     <- A
      /|\
     / | \
    /  |  \
   /   |   \
  /    |    \
  |    |    |
task task task
  1    2    3
  |    |    |
  |    |    |
  \    |    /
   \   |   /
    \  |  /
     \ | /
      \|/
       |     <- B
       |
common execution
```

Figure 10
Parallelism using Concurrent Subtasks

```
                   |
                   |
                  /|\
                 / | \
        ---------  |  ---------
       /           |           \
      /            |            \
     |             |             |
T1 := F1(X)   T2 := F2(Y)   T3 := F3(Z)
     |             |             |
     |             |             |
      \            |            /
       \           |           /
        ---------  |  ---------
                 \ | /
                  \|/
                   |
         RESULT := T1 + T2 + T3
```

Figure 11
Parallelism in Arithmetic Expressions

This general structure may be conveniently represented syntactically through the use of the statement forms <u>cobegin</u> and <u>coend</u>.

```
cobegin
  TASK1:
    -- statements in TASK 1 --
  TASK2:
    -- statements in TASK 2 --
  TASK3:
    -- statements in TASK 3 --
coend;
```

3.1.2 Lexical Definition of Tasks

In contrast to the example of parallelism outlined above, many practical applications involve a form of concurrency in which the individual processes are considerably more independent in their operation. In many multiprocessing applications, it is convenient to use independent processes to model the activity of a set of relatively independent components that together make up a complete system.

In order to illustrate the nature of parallelism arising in this class of applications, it is useful to consider the example of a terminal concentrator system whose function is to connect a large number of terminals to some small number of host computers. Clearly this type of system has a high potential for parallelism since the traffic to each terminal and to the individual host computers may presumably proceed concurrently. One very convenient structure for a system of this kind is to associate each terminal with a process (or perhaps a small set of related processes) responsible for controlling the activity for that terminal and, similarly, assign an independent process to each host computer. Given this approach, the structure of the component processes reflects the logical structure of the system and encourages a highly modular software design.

In this class of applications, it is inconvenient to think of these independent processes using the cobegin/coend model. The syntax of the cobegin/coend structure emphasizes the division of a single path into parallel subcomponents. In this case, understanding the actual mechanism used to initiate and terminate the individual processes is peripheral to understanding the dynamics of the system in operation. In these applications, it is particularly valuable for the syntactic structure to reflect the logical integrity of the component processes viewed from the perspective of the fully operational system. From this point of view, most processes are thought of as perpetual in the sense that process activation and termination are "unusual" conditions that need not be considered when examining the steady-state operation of the system.

This general notion gives rise to a representational structure for parallel control in which each component process is viewed as a logically independent entity which is therefore specified syntactically as an independent process module or task. Typically, the code controlling the initiation of a new process is not specified as part of the task but is coded separately as part of the system initialization code through the use of a FORK or INITIATE construct.

Using this structure in our model of the terminal concentrator gives rise to the following sort of representation, illustrated here using the Ada language. The terminal tasks are defined as lexically integral units. The specification

```
task TERMINAL (1 .. 100)
  -- specification of the interface between the  --
  -- terminal tasks and the external environment --
end TERMINAL;

task body TERMINAL (1 .. 100) is
  -- local declarations --
begin
  -- code to handle the terminal and any requests --
  -- from external tasks                          --
end TERMINAL;
```

defines an array of 100 terminal processes each of which will be used to control the activity of a single terminal. The terminal processes are activated in Ada through the use of an INITIATE statement which can activate all of the terminal processes simultaneously, as in

```
initiate TERMINAL (1 .. 100);
```

or individually, as in

```
initiate TERMINAL (I);
```

The second form above is useful if it is desirable to activate terminal processes dynamically rather than through static allocation.

This form of process activation has several advantages over the cobegin/coend construct for certain applications:

(1) The static definition of a task separates the description of the process from the code body which initiates the process in a way that encourages program modularity. This is less of an advantage if there is some important relationship between the initiating statements and the body of the task, as is true in the example of parallel evaluation of expressions used to illustrate the cobegin/coend structure above. In practice, however, the initiation of a process tends to be performed at system initialization time and the activity of the initiated process is completely unrelated to that of the initiating routine.

(2) The cobegin/coend sequence provides a clear structure for process initiation but makes it more difficult to develop an adequate process communication facility since it effectively obscures the identity of the independent processes. This will become more clear in the discussion of process control in the following section.

(3) The cobegin/coend structure, as presented, does not provide any effective mechanism for generating a large family of processes as required in the terminal concentrator example. Even if an extended mechanism were defined to support families within the cobegin/coend structure, the fact that task definition is not

considered as a fundamentally distinct notion from process initiation makes it unlikely that any acceptable mechanism could be proposed that would allow for dynamic activation of single members of the task family.

3.2 Classical Mechanisms for Process Control

3.2.1 Process Interaction

Parallel processes which operate on disjoint sets of variables are called disjoint or noninteracting processes. With disjoint processes it is theoretically possible to achieve the full power of parallelism, since it is never necessary for one process to wait for another. Disjoint processes can therefore be analyzed as a collection of independent sequential programs. Unfortunately, the usefulness of disjoint processes is limited; in most applications, processes must access and change common variables. In order to exploit the full power of parallelism in these cases, it is important to understand and be able to control process interactions. In this section we describe the three primary reasons for process interactions: mutual exclusion, synchronization, and communication. In the remaining sections of this chapter we discuss language features which have been proposed for controlling such interactions.

Mutual Exclusion: Numerous situations exist in which parallel processes must reference and update shared data structures. Unless a process has exclusive access to the data structure while it is making the update, the data structure may be left in an inconsistent state. Consider, for example, a data base which can be accessed by two types of processes: <u>reader processes</u> and <u>writer processes</u>. The reader processes correspond to queries from data base users and do not change the state of the data base. The writer processes, on the other hand, correspond to updates and do change the state of the data base. While reader processes can be allowed to access the data base simultaneously, it is clear that writer processes must have exclusive access to the data base while making a change.

We use the term <u>critical region</u> for a section of code in which a process needs exclusive access to some data structure. Any implementation of critical regions must satisfy the following conditions:

(1) At most one process can be inside a critical region at a time.

(2) A process waiting to enter a critical region must be allowed to do so in a finite amount of time.

(3) A process must remain in a critical region for only a finite amount of time.

<u>Synchronization</u>: Process synchronization is needed to insure that a process does not proceed past a given point without receiving an explicit signal. For example, consider a real time system which is used to monitor parallel physical activities in

an oil refinery. It is convenient to model the operation of the refinery by setting up a separate process to monitor each of the component activities. Thus one process might monitor the production of gasoline, and another process might monitor the availability of tank cars to transport the gasoline. These processes can run independently until the physical operations merge, i.e. until it is necessary to fill the tank cars with gasoline. At that point the processes must be synchronized to ensure proper operation of the physical system (don't open the valve until there is a car under the hose). Numerous other examples of process synchronization occur in operating systems where process schedulers, I/O drivers, command interpreters and so on must all be properly coordinated. Note that the signal on which the synchronization of a process depends can come from another process or from external hardware connected to the computer system.

Communication: In multi-process systems it is frequently necessary for executing processes to exchange information. Although interprocess communication can always be implemented by means of critical regions and shared data, in many cases it is more convenient to provide processes with a facility for the explicit exchange of messages. A typical example of such a situation might be an operating system process which is responsible for reformatting output data and sending it to disk storage. A natural way of organizing the code for the reformatting process is to have user processes send it messages which are appropriately transformed and sent on to the process which handles disk I/O. We use the term mailbox to refer to the location where messages are placed by a sending process and retrieved by a receiving process. Note that any implementation of message passing must provide some means for processes to agree on the size and location of mailboxes and also on how to determine when a mailbox is full.

3.2.2 Interlocks

Interlocks are perhaps the simplest and most basic technique for process control in a multi-processing system. An interlock is a special type of variable which has two distinct values: LOCKED and UNLOCKED. Operations are provided by hardware for locking, unlocking, and determining the status of these variables. The operations must be uninterruptible in order prevent an intervening access by another process; otherwise, the result returned by the operation might not reflect the true status of the interlock. Assume that we have an uninterruptible TESTANDSET instruction such as the one defined by the code segment below: *

----
* Throughout the body of this paper, all code is written using the Ada Language [Ichbiah79], primarily because Ada is used in Sections 5 and 6 as a case study of the applicability of high-level constructs to the requirements of practical

```
type INTERLOCK is (LOCKED, UNLOCKED);

function TESTANDSET (L : inout INTERLOCK)
  return BOOLEAN is
begin
  if L = LOCKED then
    return TRUE;
  end if;
  L := LOCKED
  return FALSE;
end;
```

is provided by hardware, then LOCK and UNLOCK operations can be implemented as follows:

```
procedure LOCK (L : inout INTERLOCK) is
begin
  while TESTANDSET (L) loop
    -- do nothing (busy wait) --
  end while;
end;

procedure UNLOCK (L : inout INTERLOCK) is
begin
  INTERLOCK := UNLOCKED;
end;
```

Because of the busy wait loop within the body of the procedure LOCK, interlocks of this sort are often referred to as _spin locks_.

Interlocks can, in turn, be used to implement higher-level process control primitives. For example, critical regions are

----

multiprocessors. Occasionally, such as in the TESTANDSET example (which uses an INOUT parameter to a function), the restrictions of Ada are relaxed to improve program clarity, although the syntax and general program structure are retained.

implemented by means of interlocks in the COL programming language [Evans77]. The syntax for critical regions in COL (taken from [BrinchHansen73]) is

region E do S end region

where E is an interlock and S is a statement. When a process attempts to execute a critical region, the interlock E is tested repeatedly until it is found to be unlocked. The variable E is then locked, and the statement S is executed. After execution of S is complete, E is unlocked and flow of control passes to the next statement in the program.

Since a process attempting to enter a critical region must wait in a loop (busy wait) until the lock is free, programmers must be careful that processor time is not wasted. In addition, interlocks cannot provide any guarantee of _fair access_ by processes to critical regions. A processor waiting to enter a critical region may theoretically be delayed indefinitely if other processors repeatedly enter a critical region protected by the same interlock first. Nevertheless, for applications where locks are of short duration (on the order of milliseconds), interlocks may be preferable to process control mechanisms which require the use of a scheduler.

3.2.3 Semaphores

In many process control problems it is necessary to count the number of units of some critical resource which are available to a process. Semaphores were proposed by Dijkstra in 1968 [Dijkstra68] as a way of exploiting this observation. A _semaphore_ is a special integer variable which can only be accessed by the primitives P (also called _wait_) and V (also called _signal_). If S is a semaphore, then the operations P(S) and V(S) have the following effect:

P(S): wait until S>0 and then subtract 1 from S

V(S): add 1 to S

The V operation can be used by a process to signal other processes that a given event has occurred. The P operation allows a process to delay itself while waiting for an event to occur. *

To establish mutual exclusion using semaphores, critical regions of code are bracketed by matched pairs of P and V operations as shown below:

----

* As noted later in this section, most implementations of semaphores make use of a process queue to insure fairness in the semaphore discipline. We have used this simplified formalization of semaphores to emphasize the mutual exclusion component of its operation.

```
    PROCESS A:  begin...P(S); CRITICAL_REGION_A; V(S); ...end
    PROCESS B:  begin...P(S); CRITICAL_REGION_B; V(S); ...end
```

If the semaphore S is initialized to 1 at the beginning of the program, then it is impossible for more than one process to be between a bracketed pair of P and V operations at a given time.

Semaphores can also be used for process synchronization. For example, to insure that some task T is always executed prior to tasks T1 and T2 the following scheme can be used:

```
    PROCESS A:  begin...T; V(S1); V(S2); ...end
    PROCESS B:  begin...P(S1); T1; ...end
    PROCESS C:  begin...P(S2); T2; ...end
```

In this case the semaphores must be initialized to 0. Processes B and C will be delayed until process A has completed task T1 and executed the corresponding V operation.

In order to avoid the potentially high inefficiency of busy waiting encountered in the interlock case and to insure fairness in the semaphore discipline, most implementations of semaphores are designed to interact with the process scheduler. In this case processes waiting on P operations are placed on a process queue associated with the semaphore. Whenever a V operation occurs, the next process waiting on the semaphore is removed from the queue and is made available for scheduling. If <u>enqueue</u> and _dequeue_ operations are fair (i.e., FIFO) then P and V operations will also be scheduled fairly. In applications in which fast response is needed, however, the time required for queue operations and context switching may prevent this implementation of semaphores from being acceptable as a mutual exclusion mechanism.

Although semaphores are easy to describe, programs using semaphores exhibit many of the structuring problems found in programs with _goto_ statements; they are hard to write, understand, prove, and maintain. Some typical difficulties include:

(a) It is possible to jump around a call of P and, therefore, accidentally access unprotected data [Ichbiah79].

(b) One can jump around a call on V and accidentally leave the semaphore busy so that the system deadlocks [Ichbiah79].

(c) It is not possible to program an alternative action if the semaphore is found to be busy [Ichbiah79].

(d) It is not possible to wait for one of several semaphores to be free [Ichbiah79].

(e) The use of semaphores forces the programmer to make very strong logical connections between otherwise independent processes: readers must be prepared to schedule writers and vice versa [BrinchHansen73].

(f) The programmer is forced to separate the request, grant, and acquisition of resources and introduce additional variables to represent the intermediate states "resource requested, but not yet granted" and "resource granted but not yet acquired" [BrinchHansen73].

3.2.4 Message Passing

Exchange of information between executing processes is one of the primary reasons for process interaction. Many multiprocessing systems implement explicit message passing primitives to facilitate intercommunication. Typically, a process executes a <u>send</u> command to pass a message to another process, and the target process accepts the message by executing a <u>receive</u> command. The semantics of <u>send</u> and <u>receive</u> may differ considerably depending on the methods used for storing messages and for forwarding them from one process to another. Many of these differences are illustrated by the message passing primitives in the RED Language [Nestor79] developed by Intermetrics and in Hoare's Communicating Sequential Processes language [Hoare77].

In RED, a <u>mailbox</u> is a special variable which can only be accessed by <u>send</u> and <u>receive</u> primitives. Mailboxes are declared by giving the type of the message that the mailbox will hold and the maximum number of messages that can be sent to the mailbox but not yet received. For example,

```
var M:  mailbox [string [ascii] (4)] (3);
```

declares M to be a mailbox capable of holding three messages each of which is an ASCII string of length four. Messages which have been sent to a mailbox but not yet received are stored in the mailbox in the order in which they arrive so that the mailbox acts like a FIFO queue. The command send(M, "MES1") is used to add the message "MES1" to the rear of the queue associated with mailbox M. Likewise, the command receive(M, V) is used to remove the first message from the queue associated with M and place it in variable V (which of course must be declared to be of type string [ascii] (4) also). When a process attempts to send a message to a full mailbox, the process is delayed until the mailbox is no longer full. A similar delay also occurs when a receive is attempted on an empty mailbox. Note that when message passing between processes is buffered in this manner, semaphores can be thought of as the degenerate case in which an empty message is sent each time a certain event occurs.

In Hoare's Communicating Sequential Processes language [Hoare77] message passing primitives have the syntax:

```
X?Y input Y from process X
X!Y output Y to  process X
```

Communication between processes occurs when one process names another as destination for output, and the second process names the first as source for input. In this case, the value to be output is copied from the first process to the second. This type of synchronization is called a rendezvous and is used as the basic synchronization primitive in Ada. Note that in contrast to the RED Language, there is no automatic buffering in Hoare's language; therefore, an output command may be delayed until the target process is ready to receive it.

Since buffered message passing can be easily implemented using Hoare's primitives, it can be argued that the rendezvous provides additional control of parallelism with little loss in flexibility. An obvious disadvantage of the rendezvous mechanism, however, is that in order for two processes to communicate, each process must know the name of the other. This problem occurs when it is necessary to construct a system of processes which can be expanded by the addition of new processes at a later date. In order for the new processes to communicate with the old processes, some method must be provided for making their names known to the old processes.

In general, message passing is well suited for synchronizing distributed processes running on different computers in a network and also for multi-computers such as the Tandem/16 where individual processors can only communicate through data packets transmitted over a local data bus. Message passing can also be implemented on multiprocessors with shared memory. In this case, however, the time required for transmission of messages (including buffering) may be quite large compared to other more direct methods of interprocess communication such as interlocks or semaphores.

3.2.5 Conditional Critical Regions

Hoare [Hoare72] and Brinch Hansen [BrinchHansen73] have proposed conditional critical regions as a high level process control mechanism for parallel programs. Logically related variables which must be accessed by more than one process are grouped together as resources. Individual processes are allowed access to a resource only within a conditional critical region of the form "with R when B do A end with" where R is the name of the resource, B is a boolean expression, and A is a statement whose execution may change the values of the shared variables associated with R. When execution of a process reaches the conditional critical region the process is delayed until no other process is using resource R and the condition B is satisfied. The statement A is then executed as an indivisible operation. For example, consider the standard solution to the "readers and writers problem with writer priority" [BrinchHansen73]. Each reader process has the form

```
READER:   repeat
            with R when AW=0 do RR:=RR+1 end with
            READ;
            with R when TRUE do RR:=RR-1 end with
          forever
```

and each writer process has the form

```
WRITER:   repeat
            with R when TRUE do AW::=AW+1 end with
            with R when RR=0 and RW=0 do RW::=1 end with
            WRITE;
            with R when TRUE do RW::=0; AW::=AW-1 end with
          forever
```

In this case the resource R consists of three shared variables AW (the number of writer processes which want to execute a write statement), RW (the number of writer processes currently executing write statements). and RR (the number of readers currently executing read statements).

Unfortunately, the standard implementation ([Hoare72], [BrinchHansen73]) of conditional critical regions can lead to code which is inefficient. The standard implementation uses two queues for each resource R: a _main queue_ RMAIN and a _wait queue_ RWAIT. When a process wishes to enter a conditional critical region for resource R, it enters the main queue RMAIN. Processes on the main queue are allowed to enter their critical regions one at a time. Once a process has entered its critical region, it inspects the variables of R to see if the entry condition B is satisfied. If B is satisfied, then the process completes its critical region by executing statement A. Otherwise, the process leaves its critical region and is put on the wait queue RWAIT.

When a process successfully executes the body of its conditional critical region and changes the values of the shared variables associated with R, it may cause some of the conditions on which processes in RWAIT are waiting to become true. Thus, all of the processes in the wait queue must be transferred to the main queue and allowed an opportunity to reevaluate their conditions.

Note that a given process may be transferred from the main queue to the wait queue and back several times before it finally gets a chance to execute its critical region. According to Brinch Hansen [BrinchHansen73] this busy waiting "is the price we pay for the conceptual simplicity achieved by using arbitrary boolean expressions as synchronizing conditions."

More efficient implementations of conditional critical regions can be obtained if parallel programs are preprocessed at compile time to obtain information about which conditional critical regions are enabled by the execution of a given conditional critical region. Preliminary research in this direction has been made by Schmid [Schmid76] who restricts the allowable form for conditional critical regions and by Clarke [Clarke79] who uses techniques from global flow analysis. At the present time, however, none of these optimization techniques has actually been implemented.

3.2.6 Monitors

A monitor is a collection of data and procedures shared by several parallel processes. Syntactically, a monitor consists of a series of data and procedure declarations followed by a statement which initializes the shared data.

```
monitor <monitor name>

    <declaration of local data>
    <procedure declaration 1>
    <procedure declaration 2>
            -
            -
            -
    <initialization>

end
```

The local data can only be accessed by the procedure bodies and the initialization statement. Calls on monitor procedures have the form

```
<monitor name>.<procedure name> (<parameter list>)
```

where <monitor name> is the name of a monitor and <procedure name> is one of its local procedures. Although monitor procedures may be called from several different processes operating in parallel, only one process is allowed entry into the monitor at a given time. Thus, if a call on a monitor procedure occurs while the monitor is busy executing a call from some other process, the second caller will be delayed until the monitor finishes with the first call.

Monitor prccedures may schedule their actions by means of <u>wait</u> and <u>signal</u> operations on condition queues. A <u>condition queue</u> is a queue of suspended processes which can be declared in the local data area of a monitor by the syntax

```
<condition name>:condition
```

When a

```
<condition name>.wait
```

operation is executed during a call to a monitor procedure, the monitor suspends processing of the call, places the name of the calling process on the condition queue, and releases the monitor's mutual exclusion, so that other calls may be accepted. An operation of the form

```
<condition name>.signal
```

causes execution of the first process in the condition queue to be resumed. When a signal operation occurs before the end of a monitor procedure, there may be competition between the signaling and the signaled processes for the next access to the monitor. In this case the signaling process is delayed until the signaled process has been served.

Explicit priorities may be associated with wait operations on condition queues using a statement of the form

```
<condition name>.wait (<priority>)
```

to handle those situations in which the normal first-in/first-out (FIFO) implementation of the queues does not provide satisfactory performance. When priorities are used, the process on the condition queue with the smallest priority will be resumed first when a signal operation is executed. The empty queue may be referred to by the literal "empty" in boolean expressions within monitor procedures. Thus, expressions of the form

```
<condition name> = empty
```

may be used to determine if a condition queue is empty.

To illustrate how monitors can be declared using the above syntax, we reproduce the alarm clock example from Hoare's original paper on monitors [Hoare72]. The alarm clock monitor allows a calling process to delay itself for a specified number N of time units or "ticks" by executing a call of the form ALARM CLOCK.WAKEME(N). The monitor also contains a procedure called TICK which is invoked by hardware at regular intervals (e.g., 10 times per second). Two local variables are accessed by the monitor procedures: NOW (an integer variable which records the current time) and WAKEUP (a condition variable on which processes wait).

```
monitor ALARM_CLOCK

  NOW    : integer;
  WAKEUP : condition;

  procedure WAKEME (N : in integer) is
    ALARM_SETTING : integer;
  begin
    ALARM_SETTING := NOW + N;
    while NOW < ALARM_SETTING
      loop
        WAKEUP.wait (ALARM_SETTING)
      end loop;
    WAKEUP.signal;
  end WAKEME;

  procedure TICK is
  begin
    NOW := NOW + 1;
    WAKEUP.signal;
  end TICK;

  NOW := 0;

end ALARM_CLOCK;
```

Note that the next candidate for wakening is awakened at every tick of the clock. This will not matter if the frequency of ticking is low enough. The signal statement at the end of procedure WAKEME is needed in case two or more processes are due to wake up at the same time.

In summary, monitors solve the mutual exclusion problem in an elegant manner by concentrating data and all of the code which accesses the data in a single process. Unfortunately, those applications which require the passing of messages between processes are not facilitated by monitors. Another disadvantage of monitors is that the signal and wait operations on condition queues suffer many of the structuring problems of semaphores.

4. APPLICATION CONSTRAINTS AND MULTIPROCESSORS

In recent years, considerable attention has been directed toward the design of effective language structures for the specification of parallel control. This research has resulted in the development of a number of distinct programming constructs which we have briefly surveyed in the previous section. For the most part, the existing research in the general area of concurrent programming has been relatively theoretical in nature and has concentrated more heavily on providing language forms which are "correct" from the linguistic point of view rather than on choosing those structures which are appropriate to the requirements of the application.

From our experience with the Pluribus, we believe that the primitives for parallel control must allow the programmer to develop highly efficient software which is appropriate to the class of application that tends to arise in a multiprocessor environment. Based on our own experimentation with process control in the Pluribus and drawing on the similar C.mmp experience at Carnegie-Mellon, we are convinced that many of the proposed structures for concurrent processing will prove to be extremely costly in terms of efficiency and that alternative mechanisms are more appropriate in practical situations. At the same time, many of the more theoretical treatments of parallel control provide mechanisms which, while elegant in form, solve problems which tend to arise very infrequently in applications. For this reason, we believe that any study of programming languages for multiprocessor systems must include a study of the nature of applications which are amenable to parallel decomposition and an examination of the concurrency involved.

4.1 The Need for Efficiency

At first glance, it is not immediately obvious that efficiency requirements for multiprocessor systems are likely to be any more stringent than such concerns in a traditional uniprocessor system. In fact, there is reason to suspect that efficiency is actually of less concern because a multiprocessor architecture offers, in theory, the possibility of increasing the overall system efficiency by adding processors.

To a certain extent, this suspicion is true. In a multiprocessor, it is possible to increase the effective computational speed through additional processors with a corresponding improvement in system response. Unfortunately, the problem that arises in this case is not that multiprocessors themselves require a high concern for efficiency but rather that the applications chosen for multiprocessors tend to be so time-critical in nature that efficiency is of paramount importance.

The reason that multiprocessor applications tend to have such strong efficiency requirements is in a very strong sense related to the observation that it is considerably more difficult to design software for a multiprocessor system than for a more traditional uniprocessor. To a large extent this increase in difficulty is related to the fact that multiprocessors represent a relatively new form of system architecture. When compared to the experience which has been assembled for single processor systems and sequential algorithms, very little is known about the problems involved in multiprocessor design and parallel programming. Furthermore, existing multiprocessor systems are generally experimental in nature and the associated instability tends to add further complexity to the software development process.

The fundamental implication of the difficulty associated with multiprocessors is quite simple: multiprocessor systems will rarely be used for practical applications unless the use of a multiprocessor is required by the constraints of the application. Multiprocessors have significant advantages over conventional uniprocessors in three distinct areas:

(1) Multiprocessors are capable of increased effective throughput because they allow independent tasks within the application to operate in parallel.

(2) Multiprocessors can be designed to include software reliability structures which exploit the inherent redundancy in the hardware to dynamically alter the system configuration in response to hardware failures.

(3) Multiprocessors can be expanded gracefully as the requirements of the application change.

Although the BBN Pluribus has demonstrated the effectiveness of software reliability [Robinson78] and modular expandability, it is unlikely that these considerations alone are sufficient to dictate the choice of a multiprocessor over a more conventional architecture. If reliability is a paramount concern for a particular application, then the level of reliability attainable using software techniques is often not sufficient to meet the application requirements. Furthermore, although modular expandability is clearly of practical value in many embedded systems with an extended life-cycle, it is not clear that the savings achieved fully balance the increased cost of software development and maintenance.

In light of the above considerations, the need to provide high effective throughput is likely to be the determining factor in choosing to use a multiprocessor system. Furthermore, the application must clearly be of a parallel nature in order to exploit this increased computational spee[illegible]ications chosen for use with multiprocessors tend, therefore, to have (1) high requirements for run-time efficiency and (2) highly parallel internal task structures.

It should be noted that these factors tend to increase the complexity of software development on multiprocessor systems. In his study of the efficiency of software production for real-time systems, Wolverton reports that real-time software is "three time more costly" to write than software in which there is no stringent constraint on the timing of events [Wolverton74]. Furthermore, if the system is considered to be highly complex (as is likely the case in a parallel environment) this factor becomes even larger.

4.2 The Impact of Efficiency on Language Design

From the previous section, it is clear that many applications which will be implemented on multiprocessors will have extremely high efficiency requirements and strict timing constraints. Our concern in this report is to study the influence of these constraints on the design of a high-level language. We believe that concern for efficiency leads to the following general conclusions:

(1) The use of constructs which have no efficient representation must not be required by the language design.

(2) If two different constructs display a significant variation in their efficiency depending on the application environment, both should be supplied in order to provide maximum flexibility and allow the programmer to achieve the required level of efficiency.

(3) Low-level facilities must be provided to achieve higher levels of efficiency than are attainable with any general mechanism.

Each of the above conclusions clearly has an impact on the simplicity of the language structure. While we recognize that it is important for the language to "have a consistent semantic structure that minimizes the number of underlying concepts," we emphasize that the Steelman requirements [DOD78] also indicate that the language "should be as small as possible _consistent with the needs of the intended applications_" [emphasis added]. We believe that real-time systems and highly parallel applications are common within the field of embedded computer systems and that programming languages designed for such systems must remain conscious of these requirements.

It is also important to note that the impact on overall efficiency from the use of an inappropriate mechanism for parallel control can be extremely high when compared to the efficiency cost generally associated with programming in a high-level language environment. While the techniques available for optimizing serial code are highly developed and quite successful in practice, relatively little is known about the problem of optimizing the global task structure and the internal synchronization process. Based on our experience with multiprocessor systems, we believe that these problems are extremely hard and well beyond the current state of software technology. This fact increases the importance of allowing greater flexibility in the task structure than might be required in the serial aspects of a language.

5. ADA AND MULTIPROCESSORS

5.1 Parallel Processing Facilities in Ada

In this chapter we describe and evaluate the parallel processing facilities in the preliminary design of the Ada programming language [Ichbiah79]. This language was developed by Cii Honeywell Bull for the U.S. Department of Defense to serve as a programming standard for embedded computer applications (i.e., command and control, communications, avionics, and shipboard applications, etc.). As a consequence of its intended application domain, the language contains facilities for parallel and real-time programming in addition to the usual control and data structuring facilities of conventional languages such as Pascal. In this report we limit our discussion of the Ada language to those features which directly affect the programming of multiprocessors. Because of the similarity between Ada and conventional programming languages such as Pascal, the reader should have no difficulty following the examples of this section in spite of the absence of a full description of the language. In Ada, comments are introduced by "--" and extend to the end of the line.

5.1.1 The General Structure of Parallel Tasks

Ada uses the term "task" to refer to the basic syntactic unit for process definition. A *task* consists of two parts: a *specification part* which describes the external behavior of the task, and a *task body* which describes its internal behavior. The specification part consists of a header which gives the name of the task and a declarative part which describes those features of the task which are visible to the outside world. Included in the declarative part are the declarations of those constants, types, subprograms, exceptions, and entries which are associated with the task and must be externally visible.

An example of a task specification is shown below:

```
task BUFFER is
  PACKET_SIZE : constant INTEGER := 256;
  type PACKET is array (1 .. PACKET_SIZE) of CHARACTER;
  entry READ (V : out PACKET);
  entry WRITE (E : in PACKET);
end BUFFER;
```

Entries are used for communication between tasks and look externally like procedures.

The task body consists of a declarative part which describes local data structures and a sequence of statements which implement the entry declarations described in the specification part. For the BUFFER example the task body is:

```
task body BUFFER is
  BUFSIZE : constant integer := 10;
  BUF     : array (1 .. BUFSIZE) of PACKET;
  IN, OUT : INTEGER range 1..BUFSIZE := 1;
  COUNT   : INTEGER range 1..BUFSIZE := 0;
begin
  -- statements for entries READ and WRITE --
end BUFFER;
```

The statements implementing the buffer operations READ and WRITE are given later in the chapter after additional features of Ada have been described.

The term "thread of control" is used to describe the execution of a task. When a thread of control enters a scope containing task declarations, the elaboration of each declaration creates a new potential thread of control. The _parent_ of a task is the task whose thread of control elaborates the task declaration. In order to cause the task body to be executed, the task name must be explicitly named in an _initiate_ statement, e.g.,

```
initiate PRODUCER,CONSUMER,BUFFER;
```

The tasks named in the initiate statement are activated and run in parallel with each other and with the parent task. Note that the parent of a task may be different from the task which initiated it, although both must have access to the task's name.

Consider:

```
task body T1 is
  task T2 is
     ...
  end T2;
  task body T2 is
     ...
  end T2;

  task T3 is
     ...
  end T3;
  task body T3 is
     ...
     initiate T2;
     ...
  end T3;

begin
  ...
  initiate T3;
end T1;
```

Here, T1 is the parent of T2, but T2 was initiated by T3 instead of T1.

Normal termination of a task occurs when control reaches the end of the task body. If the terminating task is a parent, then it may have to be delayed until all of its offspring have terminated. Tasks may also be terminated by means of an explicit abort statement. For example, the statement

```
abort T1,T2;
```

causes tasks T1 and T2 plus any descendant tasks to be terminated unconditionally. In this case a TASKING_ERROR exception is raised in those tasks which were communicating with the aborted task or its descendants.

Facilities are also provided for determining the status of a task. The system attribute T'PRIORITY may be used to determine the priority that has been assigned to task T by the scheduling algorithm which allocates available processors to tasks. The priority of a task may be changed by means of a call on the procedure SET_PRIORITY to reflect a change in the urgency of process execution.

Ada also provides arrays of tasks called task families to handle those situations in which it is necessary to construct a large number of similar tasks. A typical use for task families occurs when there are multiple copies of some physical device such as a console terminal and a distinct copy of the same task is necessary to drive each device, i.e.,

```
task TELETYPE_DRIVER (1..100) is
  type LINE is array (1..132) of CHARACTER;
  entry WRITELINE (TEXT : in LINE);
  entry READLINE (TEXT : out LINE);
end TELETYPE_DRIVER;

task body TELETYPE_DRIVER is
  -- statements to implement WRITELINE and READLINE --
end TELETYPE_DRIVER;
```

Individual copies of the task may be referred to by appending the appropriate subscript to the task name. Thus the statement

```
        initiate TELETYPE_DRIVER (3);
```

will cause the third copy of task TELETYPE_DRIVER to become active.

Storage for tasks may be allocated either when the task declaration is elaborated (static creation) or when the task is initiated (dynamic creation). The choice between static allocation and dynamic allocation is determined at compile time by the use of a pragma or translator command, e.g.,

```
        pragma CREATION (STATIC);
        pragma CREATION (DYNAMIC);
```

Dynamic creation is particularly important for task families where the index range provides an upper bound on the number of active processes and storage might be wasted if all tasks were allocated at the same time.

5.1.2 Entry Declarations and the ACCEPT Statement

Communication between tasks is provided by entry calls and accept statements. When one task needs to communicate with another task, it executes an entry call. Entry calls specify the information to be exchanged between the tasks and have exactly the same form as procedure calls. Thus in the bounded buffer example from the last section, a producer task places data in the buffer by executing the entry call

```
        BUFFER.WRITE (PRODUCER_DATA);
```

and a consumer task executes the call

```
        BUFFER.READ (CONSUMER_DATA);
```

to retrieve data from the buffer.

In order for an entry call to be syntactically correct, the called task must contain an _entry declaration_ with a corresponding name and formal part. Entry declarations resemble procedure declarations and contain information about the type and mode of the formal parameters of the entry. An entry declaration can also specify an array or family of entries all of which have the same name and parameters. In this case, subscripts must be used to distinguish a particular entry in the family. Thus, in a disk head scheduler it may be convenient to associate a distinct entry with each track on the disk

```
        entry TRANSFER (1..200) (D:DATA)
```

When another task wishes to write on track I, it issues an entry call of the form

```
        TRANSFER (I) (DATA_REQUEST);
```

The _accept_ statement is analogous to the body of a procedure and indicates to the called task which statements should be executed when a particular entry call occurs. The formal part of the entry declaration is repeated at the beginning of the accept statement in order to emphasize the scope of the entry parameters. Following the formal part are the statements to be executed when the entry call is accepted. The accept statements for the entries READ and WRITE in the bounded buffer example are shown below:

```
accept WRITE (E : in PACKET) do
  BUF (INX) := E;
end WRITE;

accept READ (V : out PACKET) do
  V := BUF (OUTX);
end READ;
```

The variables INX and OUTX are integers which point, respectively, to the rear and the front of the buffer and are declared in the body of the task (the complete example is presented later in this section). It is important to note that these variables need not be incremented within the accept statements. Since accept statements are executed in mutual exclusion, it is important for them to be as short as possible and not contain unnecessary statements. Accept statements for entry families must be subscripted to distinguish different entries in the same family. Thus, accept statements for the disk head scheduler example will typically have the form

```
accept TRANSFER (D : in DATA ) do ... end TRANSFER;
```

The synchronization between the calling task and the called task in an entry call is similar to the *rendezvous* that occurs with Hoare's CSP language. As in Hoare's language there are two possibilities for a rendezvous, depending on whether the calling task issues the entry call before or after the corresponding accept statement is reached by the called task. In either case the process which reaches the rendezvous first is delayed until the other process has an opportunity to catch up. When the rendezvous is achieved, the *in* parameters of the entry call are passed to the called task. The calling task is then suspended while the called task executes the body of the accept statement. After execution of the accept statement, the values of *out* parameters are passed back to the calling task, and the two tasks are allowed to proceed independently again. A queue of waiting tasks is associated with each entry to handle those situations in which several different tasks simultaneously access the same entry. Tasks are removed from the queues in a FIFO manner each time that a rendezvous occurs. Note that the naming problem which occurs in Hoare's language is avoided by Ada since it is unnecessary for a called process to know the name of the calling process.

5.1.3 The Select Statement

Many of the disadvantages of semaphores stem from lack of control over what happens when a semaphore is found to be busy. Thus, it is not possible to program an alternative action to be executed when a semaphore is busy nor is it possible to wait for one of several semaphores to be free. The select statement in Ada provides a mechanism for avoiding this type of problem. Syntactically, the select statement resembles a case statement in which each alternative is a conditional statement:

```
select
  when  B1  =>  A1;
  or when  B2  =>  A2;
  ...
  or when  BN  =>  AN;
  else S;
end select
```

Each when condition may contain an arbitrary boolean expression involving variables which are visible to the task and may be omitted if the condition is known to be true. The select alternatives A1, ..., AN are sequences of statements in which the first statement is always an accept statement or a delay statement. The else clause is simply a sequence of statements and can also be omitted if the guarding conditions B1, ..., BN are mutually exhaustive. A select alternative is said to be open if there is no preceding when clause or if the corresponding condition is true; otherwise it is said to be closed.

The execution of a select statement is described by the following five rules.

(1) All of the conditions are evaluated to determine which alternatives are open.

(2) An open alternative starting with an accept statement may be executed if the corresponding rendezvous is possible.

(3) An open alternative starting with a delay statement may be executed if no other alternative has been selected before the specified time interval has elapsed.

(4) If no alternative statement can be immediately selected and there is an else clause, then the else clause is executed next. If there is no else clause, the task waits until an open alternative can be selected by rule 2 or rule 3.

(5) If all alternatives are closed and there is an else clause, the else part is executed. If there is no else clause, the exception SELECT_ERROR is raised.

With the select statement we can now complete the task body in the bounded buffer example:

```
task body BUFFER is
  SIZE      : constant INTEGER := 10;
  BUF       : array (1..SIZE) of PACKET;
  INX, OUTX : INTEGER range 1..SIZE := 1;
  COUNT     : INTEGER range 0..SIZE := 0;
begin
  loop
    select
      when COUNT < SIZE =>
        accept WRITE (E: in PACKET) do
          BUF (INX) := E;
        end WRITE;
        INX := INX mod SIZE + 1;
        COUNT := COUNT + 1;
      or when COUNT > 0 =>
        accept READ (V: out PACKET) do
          V := BUF (OUTX);
        end READ;
        OUTX := OUTX mod SIZE + 1;
        COUNT := COUNT - 1;
    end select;
  end loop;
end BUFFER;
```

The buffer is represented by a circular array with the variables INX and OUTX indicating the portion of the array which contains data. The guard COUNT < SIZE in the first alternative of the select statement protects the buffer from overflow during the execution of a write operation. Similarly, the guard COUNT > 0 in the second alternative protects the buffer from underflow during a read operation. Note that if 0 < COUNT < SIZE and both a read call and a write call occur, the accept statement that is selected will be chosen in a completely random manner. The programmer, therefore, must be careful that this nondeterminism in the selection of alternatives does not affect the correctness of the program.

5.1.4 The Delay Statement, Interrupts and Generic Tasks

In this section we describe three additional process control features provided by Ada. These features do not affect the expressive power of the language as significantly as the features discussed previously and are therefore not described in as great detail.

The first feature is the delay statement which can be used to postpone execution of a task for a specified interval of time. The delay statement has the form

```
delay <simple expression>
```

The expression following the delay statement represents the length of time (in units of the real time clock) that the process is to be delayed. A delay statement can be used in place of an accept statement in an alternative of a select statement. In this case if no rendezvous occurs during the specified time interval, the statement list following the delay statement will be executed. Thus, an additional alternative of the form

```
or delay 10.0*MINUTES ; initiate SYSTEM_TEST;
```

may be added to the select statement in the task body for the bounded buffer example. This modification will cause the diagnostic task SYSTEM_TEST to be run if a ten minute time interval passes in which there are no READ or WRITE entry calls.

The second feature is the interrupt entry: in Ada, hardware interrupts are simply interpreted as external entry calls. An Ada representation specification is used to link the entry to the physical storage address which records the interrupt. The interrupt is processed exactly the same way that any other entry call is processed; thus, the queuing mechanism for entry calls can be used to handle multiple interrupts. Likewise, the mechanism for masking interrupts can be hidden from users by incorporating it in the software which connects the interrupts to the entry call. To illustrate how interrupts are handled in Ada, we show how a stop button can be added to the bounded buffer example. We assume the existence of a console button which can be pressed to cause a hardware interrupt. A representation specification of the form

```
for STOP use at 8#7777;
```

can be used to associate the entry STOP with the physical address of the interrupt. If the select statement in the task body is modified to include the alternative

```
or accept STOP; exit;
```

then loop will be terminated when the stop button is pressed.

The final process control feature that we discuss is the generic task. The bounded buffer example described earlier in this chapter does not provide users with a general mechanism for declaring buffer tasks. By making the tasks generic, i.e., by changing the specification part of the task to

```
generic task BUFFER is
  PACKET_SIZE : constant INTEGER := 256;
  type PACKET is array (1..PACKET_SIZE) of CHARACTER;
  entry READ (V : out PACKET);
  entry WRITE (E : in PACKET);
end
```

this difficulty can be overcome. When a user needs to declare a new instance of a bounded buffer, the construction

```
task BB is new BUFFER:
```

may be used. READ and WRITE calls on the new instance of the bounded buffer have the syntax:

```
BB.WRITE (PRODUCER_DATA);

BB.READ (CONSUMER_DATA);
```

Signals and semaphores are provided by Ada as predefined generic tasks. If Ada is implemented on a machine on which these primitives are provided by hardware, then the compiler can directly translate entry calls into the corresponding hardware primitives. In doing so, however, it is critical that the semantics of the language remain entirely unchanged. As noted in Section 6.2.3, the FIFO semantics of the ADA rendezvous can make this particularly difficult to achieve.

AD-A107 533   BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA   F/G 9/2
THE IMPACT OF MULTIPROCESSOR TECHNOLOGY ON HIGH-LEVEL LANGUAGE --ETC(U)
SEP 79   A EVANS, C R MORGAN, E S ROBERTS   DCA100-78-C-0028

UNCLASSIFIED   BBN-4188   NL

2 of 2
AD
A107533

END
DATE
FILMED
12-81
DTIC

6. EVALUATION OF PROCESS CONTROL IN ADA

As discussed in Section 4, we believe that the use of multiprocessor systems tends to be most valuable in those applications in which run-time efficiency is a critical concern. For this reason, we feel that the parallel control features provided by an implementation language intended for use with multiprocessors must be designed to allow highly efficient implementation of interprocess communication and control. After reviewing the Ada language in detail, we are concerned that the primitives provided by Ada for process control will not allow the programmer to achieve this desired level of efficiency or will force an unnatural coding discipline that will permit some gain in efficiency at the cost of making programs more difficult to read and understand.

6.1 Scheduling and the Rendezvous

The most severe problem with the process control features in Ada from the point of view of efficiency is that the transmission of data from a sender process to a receiving process requires excessive scheduler interactions. Our experience is that message passing of this type occurs frequently in real-time applications, and that in such applications it is necessary to reduce the number of interactions with the scheduler to a minimum to meet the relevant time constraints.

6.1.1 An Example of Scheduling Delay

To illustrate this problem, we examine the problem of passing messages from a sender process to a receiving process where no response or acknowledgment is required. Conceptually, we imagine that there is a queue linking the sender and receiver which can hold some finite number of messages in transit. When the sender process generates a message, it enters the associated data at the end of the queue. The receiver process, whenever it is free to accept a new message, simply takes the first message from the queue. In a parallel environment, it is desirable that the sending operation (i.e., entering the data on the queue) be performed without incurring any significant delay so that the sending process can continue its operation as quickly as possible. In particular, in the usual case in which the queue is not full, there should be no required scheduler interactions.

Consider the bounded buffer example presented in Section 5.1 (the code is reproduced below for easier reference within this section). This example has been used to demonstrate that buffered message passing with nonblocking senders can be implemented in Ada. If entry calls are implemented as described in the Ada Rationale [Ichbiah79, page 11-40], however, the delay arising from scheduler actions seems extremely severe and impossible to avoid.

```
task BUFFER is
  PACKET_SIZE : constant INTEGER := 256;
  type PACKET is array (1 .. PACKET_SIZE) of CHARACTER;
  entry READ (V : out PACKET);
  entry WRITE (E : in PACKET);
end BUFFER;


task body BUFFER is
  SIZE       : constant INTEGER := 10;
  BUF        : array (1..SIZE) of PACKET;
  INX, OUTX  : INTEGER range 1..SIZE := 1;
  COUNT      : INTEGER range 0..SIZE := 0;
begin
  loop
    select
      when COUNT < SIZE =>
        accept WRITE (E: in PACKET) do
          BUF (INX) := E;
        end WRITE;
        INX := INX mod SIZE + 1;
        COUNT := COUNT + 1;
      or when COUNT > 0 =>
        accept READ (V: out PACKET) do
          V := BUF (OUTX);
        end READ;
        OUTX := OUTX mod SIZE + 1;
        COUNT := COUNT - 1;
    end select;
  end loop;
end BUFFER;
```

Consider, for example, the scheduler interactions involved when a producer task sends a packet of data to a consumer task. Assume that the producer task executes the entry call

```
BUFFER.WRITE (PRODUCER_DATA);
```

to initiate the transfer. Given the semantics of the entry call, the producer is now blocked until the buffer task is scheduled and completes the rendezvous. During this time, the producer process is suspended and must wait to be rescheduled when the buffer task completes. Thus, before the producer is allowed to continue, two scheduling operations must occur. Furthermore, the implementation discussion in the Ada Rationale indicates that the buffer task should dismiss after completing the rendezvous in order to allow tasks of higher priority to run at that point, so that it will not immediately be able to perform a rendezvous with a consumer process.

Essentially the same sequence of operations is performed when the consumer task executes the corresponding entry call

```
BUFFER.READ (CONSUMER_DATA);
```

to receive a message. This implies that a total of four scheduling interactions are required to transmit a single message. Since each scheduler interaction may involve a complete context swap, this implementation of message passing would be prohibitively expensive for many applications.

Note that this problem does not arise if the message passing mechanism is implemented through the use of a message queue or directly by the hardware of the target machine. The queue operations themselves must be protected against concurrent updates through some mutual exclusion mechanism, but in this case it is reasonable to use interlocks or some similar mechanism based on busy waiting without incurring the overhead of a scheduler interaction. From the statistics on lock contention given in [Oleinick78] which is reproduced in Section 6.2. we see that neither the producer task nor the consumer task will be delayed for an inordinate period of time.

From our experience with real-time communications systems, it is evident that the scheduling delay outlined above presents a serious problem that must be solved for Ada to be recognized as an acceptable implementation language for multiprocessor systems. In the search for a solution, one has two potential choices:

(1) Add new features to Ada to support a more efficient mechanism for message passing without sender delays.

(2) Without changing the Ada language, develop some mechanism which would permit the translator to produce more efficient code in those cases where it can be determined that the rendezvous is not necessary.

Of the two approaches, the second has a number of distinct advantages, assuming that this type of optimization is possible in any interesting class of problems. We believe, however, after considering a variety of suggestions designed to support the elimination of rendezvous delay at translation time, that any complete solution will prove to be extremely complex and largely unworkable in practice. To illustrate the complexity of the problem and the difficulties that arise in existing attempts at a workable solution, we will consider the following potential technique, originally presented by Habermann in his commentary on the RED and GREEN candidates for the Ada language [Lamb79].

6.1.2 The Habermann Implementation of Rendezvous

Briefly presented, the Habermann approach consists of replacing (in terms of the underlying implementation) the entry/accept interface with one that more closely resembles a procedure call linkage. The interesting feature of this change in implementation is that the statements in the range of the accept statement are evaluated, not by the called task, but rather by the caller. If this is done correctly, the calling task need never dismiss its processor and therefore is not forced to wait for the scheduler.

In his evaluation of the Ada tasking facility, Habermann observes that many of the tasks that arise in practical applications may be considered to be of the "server" type and consist of one or more select statements enclosed in a loop (the [illegible]ER task above is of this type). Habermann argues that tasks of this type often permit the compiler to eliminate the rendezvous by replacing the accept statement linkage with a subroutine which implements the required mutual exclusion and synchronization with some internal primitive such as a semaphore. He briefly outlines a scheme for performing this transformation by analyzing a variety of cases. In the paragraphs below, we have attempted to reconstruct this argument in a simpler form and then apply it to the BUFFER example.

As a simple case, consider a task whose body consists entirely of a sequence of accept statements in a loop (note that this task has the same structure as the generic task SEMAPHORE which is predefined in Ada), such as

```
task body EXAMPLE1 is
begin
  loop
    accept ENTRY1 do
      -- <body of ENTRY1> --
    end ENTRY1;
    accept ENTRY2 do
      -- <body of ENTRY2> --
    end ENTRY2;

    --  more accept statements  --

    accept ENTRYn do
      -- <body of ENTRYn> --
    end ENTRYn;
  end loop;
end EXAMPLE1;
```

To translate this example into its procedural equivalent, we associate each of the entries (ENTRYi) with an internal semaphore (SEMi) and translate each accept statement into a procedure declaration which begins by performing a P operation on its associated semaphore and ends by performing a V operation on the semaphore associated with its successor entry (modulo n). The "entry procedures" then have the form

```
procedure ENTRY1 is
begin
  SEM1.P;
  -- <body of ENTRY1> --
  SEM2.V;
end ENTRY1;
```

and so on up to

```
procedure ENTRYn is
begin
  SEMn.P;
  -- <body of ENTRYn> --
  SEM1.V;
end ENTRYn;
```

In this case, since no code exists in EXAMPLE1 that is not enclosed in accept statements, no actual thread of control need exist for EXAMPLE1 and the initiation of EXAMPLE1 consists simply of setting the state of SEM1 to UNLOCKED and the remaining semaphores to LOCKED. After considering the actions of the semaphores in the example above, it should be clear that the control semantics of the procedural version is identical to that of the rendezvous provided that semaphores are implemented so as to insure the first-in/first-out discipline. At the beginning, the EXAMPLE1 "task" will only accept entry calls to ENTRY1, since any other call will block on the P operation at entry. The first call to ENTRY1, on the other hand, will succeed, and the V operation at the end of the procedure body will allow the system to accept a call on ENTRY2 or to process an existing call pending on the associated semaphore.

The select statement may also be handled through the use of semaphores in a similar fashion. Consider, for example, the task specification below:

```
task body EXAMPLE2 is
begin
  loop
    select
      accept CASE1 do
        -- <body of CASE1> --
      end CASE1;
    or
      accept CASE2 do
        -- <body of CASE2> --
      end CASE2;
    end select;
  end loop;
end EXAMPLE2;
```

In this example, we will need to associate a semaphore with the select statement (SELECT_SEM) to insure mutual exclusion of the independent entries (in the general case, other semaphores must be associated with the accept statements themselves to provide synchronization which is unnecessary here). This task may be coded in procedure form as follows:

```
procedure CASE1 is
begin
  SELECT_SEM.P;
  -- <body of CASE1> --
  SELECT_SEM.V;
end CASE1;
```

```
procedure CASE2 is
begin
  SELECT_SEM.P;
  -- <body of CASE2> --
  SELECT_SEM.V;
end CASE2;
```

Once again, initiation of the task EXAMPLE2 corresponds to setting the state of SELECT_SEM to UNLOCKED thus allowing the first entry call to succeed. In this example, the first call on either of the entries CASE1 or CASE2 will succeed and will perform the actions in the body of the associated accept statement range in mutual exclusion of all other operations because of the protection provided by the semaphore. Upon completion of the entry body, the semaphore will once again become free and the system may service any further calls on either of the entries. It is interesting to note that this program transformation provides for "random" ordering in the select statement by implicitly implementing the "order of arrival" method discussed in the Ada Rationale.

This treatment of the select statement, however, does not include the use of the when clause to guard a particular alternative in the select body. Fortunately, this does not affect the nature of the solution dramatically, because the effect of the when clause can be incorporated into the entry procedures associated with that alternative. For example, if the first alternative in the above select statement had been written as

```
select
  when BOOLEAN_GUARD_EXPRESSION =>
    accept CASE1 do
      -- <body of CASE1> --
    end CASE1;
  . . .
end select;
```

the corresponding entry procedure could be coded as

```
procedure CASE1 is
begin
  loop
    SELECT_SEM.P;
    exit when BOOLEAN_GUARD_EXPRESSION;
    SELECT_SEM.V;
    delay APPROPRIATE_SCHEDULING_INTERVAL;
  end loop;
  -- <body of CASE1> --
  SELECT_SEM.V;
end CASE1;
```

The examples presented above, however, are overly simplified in that they do not provide for code within the body of the task which is not enclosed in an accept statement. This case requires a slightly more complex treatment that forces the server task to maintain an independent thread of control. To illustrate the basic notion involved in this generalization, consider the simple task skeleton below:

```
task body EXAMPLE3 is
begin
  loop
    -- <statement body 1> --
    accept ENTRY1 do
      -- <body of ENTRY1> --
    end ENTRY1;
    -- <statement body 2> --
    accept ENTRY2 do
      -- <body of ENTRY2> --
    end ENTRY2;
  end loop;
end EXAMPLE3;
```

With the exception of the intervening <statement body> code, this task is identical in form to that given in task EXAMPLE1, and we would like to identify some similar procedural form for the bodies of the entry calls. This can be done by associating each of the <statement body i> segments with a semaphore (STATEMENT_SEMi) in much the same way as the entry semaphore association (here ENTRYi is associated with the semaphore ENTRY_SEMAPHOREi). The task is then divided into a component which represents the "real" task (i.e., the code outside of the accept statements) and the entry procedures, giving rise to the code segments below:

```
task body TRANSFORMED_EXAMPLE3 is
begin
  loop
    STATEMENT_SEM1.P;
    -- <statement body 1> --
    ENTRY_SEM1.V;
    STATEMENT_SEM2.P;
    -- <statement body 2> --
    ENTRY_SEM2.V;
  end loop;
end TRANSFORMED_EXAMPLE3;
```

```
procedure ENTRY1 is
begin
  ENTRY_SEM1.P;
  -- <body of ENTRY1> --
  STATEMENT_SEM2.V;
end ENTRY1;
```

and so on up to

```
procedure ENTRY2 is
begin
  ENTRY_SEM2.P;
  -- <body of ENTRY2> --
  STATEMENT_SEM1.V;
end ENTRY2;
```

In this example, each of the statement sequences enables the succeeding entry and vice versa, giving rise to the correct semantics with respect to synchronization and mutual exclusion.

To illustrate the actual implications of this approach, we now consider the transformation of the BUFFER task according to the synthesis of these individual transformations. For simplicity, all statements within the range of a select alternative have been moved inside the corresponding accept statement, although the technique used in EXAMPLE3 illustrates the general method for restoring the available potential concurrency.

```
package NEWBUFFER is
  PACKET_SIZE : constant INTEGER := 256;
  type PACKET is array (1 .. PACKET_SIZE) of CHARACTER;
  procedure READ (V : out PACKET);
  procedure WRITE (E : in PACKET);
end NEWBUFFER;


package body NEWBUFFER is
  SIZE      : constant INTEGER := 10;
  BUF       : array (1..SIZE) of PACKET;
  INX, OUTX : INTEGER range 1..SIZE := 1;
  COUNT     : INTEGER range 0..SIZE := 0;

  procedure WRITE (E: in PACKET) is
  begin
    loop
      BUFFER_SEM.P;
      exit when COUNT < SIZE =>
      BUFFER_SEM.V;
      delay APPROPRIATE_SCHEDULING_INTERVAL;
    end loop;
    BUF (INX) := E;
    INX := INX mod SIZE + 1;
    COUNT := COUNT + 1;
    BUFFER_SEM.V;
  end WRITE;

  procedure READ (V: out PACKET) is
  begin
    loop
      BUFFER_SEM.P;
      exit when COUNT < SIZE =>
      BUFFER_SEM.V;
      delay APPROPRIATE_SCHEDULING_INTERVAL;
    end loop;
    V := BUF (OUTX);
    OUTX := OUTX mod SIZE + 1;
    COUNT := COUNT - 1;
    BUFFER_SEM.V;
  end READ;

end NEWBUFFER;
```

From the point of view of efficiency, it is evident that the above implementation strategy is preferable to the cooperating process model of rendezvous suggested in the _Ada Rationale_, but there are clearly some costs associated with this approach, largely in terms of the complexity this structure imposes on an otherwise simple model. In particular, it is important to recognize that the Ada semantics cannot be maintained if the body of the accept statement is viewed purely as a subroutine of the caller which communicates with the called task solely through the internal semaphore structure. The generated code must take account of the fact that two separate tasks are involved.

The complexity arises because of the "identity crisis" which occurs for the the task executing the statements within an accept body. In many ways, it is convenient to think of the calling and called tasks as completely distinct entities to make the relationship between the separate threads of control as distinct as possible. This view is made explicit in the _Ada Rationale_ (page 11-40) which emphasizes that "the caller executes a procedure himself whereas an accept statement is executed by the callee on the caller's behalf." Under the Habermann implementation, this distinction is no longer clear since the fundamental savings in efficiency results from having the calling task execute the accept body in much the same manner as a procedure call.

In some cases, the identity of the task executing the code may be of some importance. For example, in order to allow metering of an application program, it is important that the runtime consumed during the accept body be charged to the CLOCK attribute of the called task rather than its caller. It is also important to remember that exception conditions which occur during the execution of the accept statement must be raised in both the caller and called task.

Considerations such as these seem to indicate that some form of context switching to identify the called task must be performed as part of the entry/accept linkage. Although this does not necessarily present fundamentally difficult problems for the resulting implementation, it is clear that the resulting scheme remains both less efficient and more complex conceptually than the basic queuing model we originally wanted to achieve.

6.1.3 Automatic Data Queuing

An alternative approach to the problem would be to devise some technique for adopting a queue implementation while retaining the linguistic structure of the entry/accept linkage. Presumably, this sort of structure is only meaningful in those cases in which the flow of information is unidirectional and where the synchronization provided by the rendezvous is known to be irrelevant. In these cases, it is possible to achieve a significant increase in message passing efficiency by building a data queue into the task communication structure and allowing the sender to proceed.

It is immediately evident that this type of approach changes the nature of the implementation strategy. In the implementation of the rendezvous proposed in the *Ada Rationale* or the Habermann alternative described above, no form of data queuing is ever supported by the implementation. The only entities which are entered in queues are tasks, and each task, because of the structure of the rendezvous, may be entered on at most one queue. This is extremely convenient since it allows arbitrary queuing of tasks without encountering a memory allocation problem; it is sufficient to reserve a queue pointer cell in the activation record of each task. Data queuing, on the other hand, requires that space be available to hold each of the data items on the queue. Assuming that dynamic allocation of this queue space is unmanageable, one is required to impose an upper bound on the queue size which is fixed at translation time.

In order to illustrate the general mechanism, consider the task specification below which performs the inverse of the LINE_TO_CHAR function illustrated in the *Ada Rationale* (page 11-6).

```
task CHAR_TO_LINE is
  type LINE is array (1 .. 80) of CHARACTER;
  entry PUT_CHAR <80> (C : in CHARACTER);
  entry GET_LINE (E : in LINE);
end RECEIVER_EXAMPLE;
```

```
task body CHAR_TO_LINE is
  BUFFER : LINE;
begin
  loop
    for I in 1 .. 80 loop
      accept PUT_CHAR (C : in CHARACTER) do
        BUFFER (I) := C
      end PUT_CHAR;
    end loop;
    accept GET_LINE (L : out LINE) do
      L := BUFFER;
    end GET_LINE;
  end loop;
end CHAR_TO_LINE;
```

Note that the syntax of the entry declaration has been extended to allow a queue size indicator as in

```
entry PUT_CHAR <80> (C : in CHARACTER);
```

The <80> parameter specifies a queue size for communication between the callers of PUT_CHAR and the CHAR_TO_LINE task itself. In this case, the first eighty calls to PUT_CHAR will simply copy their data into the character queue established by the entry declaration and proceed, even if the CHAR_TO_LINE task is unable to complete the rendezvous for the PUT_CHAR entry (presumably because it is waiting for a call to GET_LINE). Thereafter, additional calls to PUT_CHAR will block and be suspended until characters are taken from the queue by the CHAR_TO_LINE task.

For the most part, the implementation of this extension to the rendezvous mechanism is completely straightforward. For the case of an entry which has only <u>in</u> parameters, the calling task performs one of two actions when making an entry call. If the queue is not full, the input parameters are copied into the pre-allocated data area and added to the end of the queue; if the queue is full, the task activation record is queued for that entry in exactly the same manner as that used in the complete rendezvous approach. The server task, upon reaching an accept statement, looks to see if the queue is empty. If so, the server task is dismissed and waits for an entry call; if there are entries in the queue, the data items from the first entry are copied into the server task. As part of the same operation, the parameters from the first task (if any) in the associated queue of sending tasks must be appended to the end of the data queue, at which point the sending task is free to proceed.

A similar mechanism can be used to handle the case of entries which operate in the opposite direction and have only <u>out</u> parameters. In this case, receiving tasks are suspended when the data queue is empty and the server must wait when the data queue is full.

This approach makes considerable sense if one argues that many applications require efficient message passing structures and that those structures should be incorporated into the language in a manner consistent with the existing mechanism for synchronization. One important observation about this approach is that the queue size information specified by the entry syntax may be interpreted in much the same fashion as a pragma statement (which may be a more appropriate syntax) which the translator is free to ignore. If some translator chooses to implement all entry calls using the complete rendezvous scheme, this will only affect the efficiency of the resultant program rather than the semantics.

6.1.4 Communication through Low-Level Facilities

One further alternative to be considered is to provide low-level facilities for mutual exclusion which would allow programmers who require more efficient message passing than that provided by the rendezvous to implement other message passing disciplines. While we do not feel that low-level facilities are required for an efficient solution to interprocess communication, we believe that there are other independent reasons which argue for the introduction of such facilities. If these are provided, it may be unnecessary for the language to supply any additional mechanisms for communication since it will be possible for the users to create additional structures to achieve the necessary level of efficiency.

6.2 Low-Level Synchronization Facilities in Ada

A related problem which limits the potential efficiency of Ada arises from the lack of low-level facilities for protecting shared data against concurrent access. In Ada, the only mechanism available for providing mutual exclusion is through the rendezvous of an entry call in one task and an accept statement in another. Although we feel that the entry/accept linkage is a powerful tool which will be useful over a wide range of applications, there are limitations in the structure which will make it difficult to use Ada in certain application environments in which efficiency is of considerable importance unless additional primitives are included so as to provide a more flexible synchronization mechanism.

6.2.1 Synchronization and Efficiency

As noted in the previous section, the rendezvous mechanism imposes a significant overhead cost which typically consists of two scheduling events for each execution of a critical region. While this cost may be reduced considerably through the use of alternative implementation strategies, the fact that mutual exclusion involves the cooperative activity of a calling task and a server task implies that, even in the best of circumstances, there will be some overhead cost involved in context switching between the two task.

The actual impact of the rendezvous overhead depends to a large extent on the frequency of access to shared data and on the size of the critical regions. Clearly, any application in which access to shared data structures is infrequent is not significantly affected by the scheduling overhead. Similarly, if the size of the critical regions is large (in terms of the amount of computation required) in comparison to the rendezvous cost, overall system performance is relatively insensitive to this delay.

On the other hand, consider the extreme case of an application in which access to shared data is frequent (such as on the order of 10% of the instructions executed not counting those required for parallel control) and yet the actual size of a typical critical region is very short (perhaps as little as one or two instructions). In this case, system throughput is largely determined by the efficiency of the mutual exclusion mechanism. If spin locks are used in this environment, it is not unreasonable to expect that a typical cycle from one critical region to the next would require on the order of twenty instructions, assuming that lock contention is not prohibitive. If scheduling interactions are required to insure mutual exclusion, the path through a critical region would be significantly more costly and would typically require more than 200 instructions, which suggests an order of magnitude reduction in overall efficiency.

While the severity of the problem is exaggerated by the example above, the ratio of synchronization time to time spent in critical regions is an important factor in many applications.

Furthermore, the choice between spin locks and scheduler-based synchronization mechanisms does have a significant impact on synchronization time. In the Hydra system, for example, spin locks are two orders of magnitude faster than the fastest synchronization primitive involving the scheduler [Oleinick78]. Since spin locks can be implemented using between three and ten instructions on most machines, this factor of 100 is likely to be representative of the relative cost for a wide range of systems.

The effect of this differential in the efficiency of the various synchronization primitives is that different applications may require different mechanisms according to the size of the critical regions involved. After studying the performance of a parallel root-finding application on C.mmp using a variety of synchronization mechanisms, Oleinick and Fuller [Oleinick78] conclude that each of the scheduling mechanisms supported by C.mmp or the Hydra operating system has an associated operating range. If the time between synchronization events is relatively short (in this case, less than about 15 milliseconds), spin locks are the only synchronization mechanism available which incurs a synchronization cost of less than 50%. If the interval between synchronization events is longer, the more powerful primitives provided by the scheduler become more appropriate and less costly. Figure 12 (which is reproduced from [Oleinick78]) illustrates the relative efficiency of five different scheduling primitives available on C.mmp as a function of the expected inter-synchronization time. In Figure 12, the curve labelled "spin lock" corresponds to synchronization managed by interlocks which do not depend on any scheduler interactions. The "kernel semaphore" curve indicates the behavior of a simple semaphore mechanism implemented by the operating system as the low-level synchronization mechanism intended for use by system processes. The remaining curves trace the behavior of several implementations of policy module (PM) semaphores, which are implemented as part of the scheduling algorithm.

The fact that different operating ranges exist suggests that some flexibility must be available in the choice of scheduling primitives in order to allow the system to meet the requirements of a particular application. The lack of this flexibility in Ada implies that the language may not be appropriate to applications in which the expected time between synchronization events is small. In our experience, this is frequently the case in real-time applications and we feel strongly that the introduction of low-level synchronization primitives into the Ada language is necessary to handle this class of applications with the required level of efficiency.

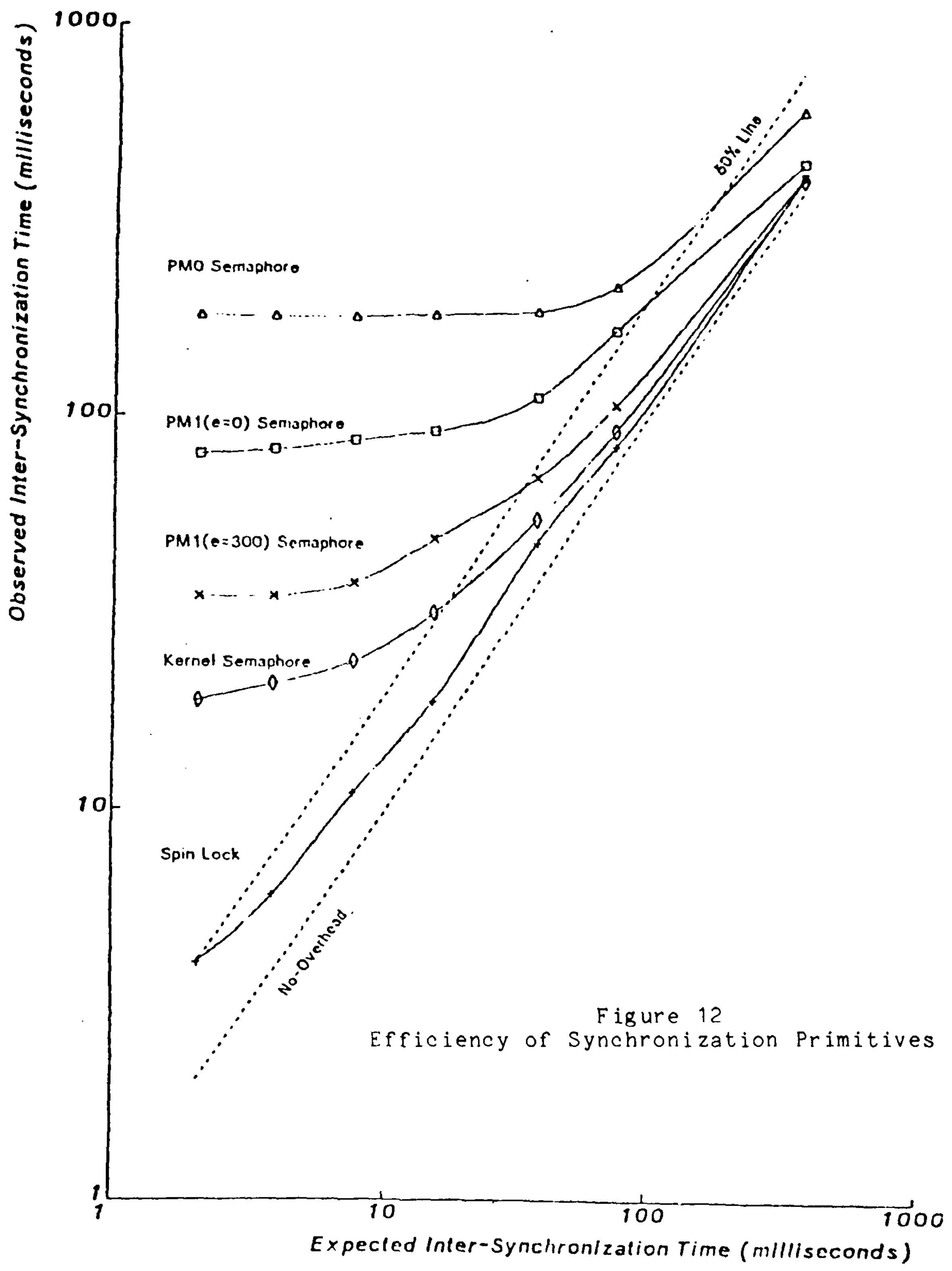

Figure 12
Efficiency of Synchronization Primitives

6.2.2 Control-Based vs. Data-Based Synchronization

In addition to the efficiency concerns discussed in the previous section, the rendezvous mechanism provided by the entry/accept linkage in Ada differs from many of the conventional notions of synchronization in the sense that mutual exclusion is a function solely of the task (or control structure) and is independent of the data structure as seen by the application program. This property appears to have an effect on memory utilization if conventional program structuring is used.

Consider an application in which some relatively large number of entities may be manipulated through the use of some moderately large number of actions (for concreteness in this example, assume that there are 100 entities and 10 actions) in such a way that mutual exclusion is required to prevent two actions from occurring simultaneously for the same entity. This type of situation occurs frequently in a wide variety of applications; for example, the terminal concentrator presented in Section 3 might well be written using a structure in which each terminal represented a distinct entity and various commands entered on the individual keyboards would trigger actions operating on that terminal.

In Ada, this situation would ordinarily be modeled through the use of a _task family_ whose members corresponded to the individual entities. The actions correspond to entries in the body of the task, which would give rise to the following general structure:

```
task ENTITY (1 .. 100) is
  entry ACTION1;
  entry ACTION2;
  -- entry declarations for remaining actions --
  entry ACTION10;
end ENTITY;


task body ENTITY is
begin
  loop
    select
       accept ACTION1 do
         -- body of action 1 --
       end ACTION1;
    or accept ACTION2 do
         -- body of action 2 --
       end ACTION2;
    -- accept statements for remaining actions --
    or accept ACTION10 do
         -- body of action 10 --
       end ACTION10;
    end select;
  end loop;
end ENTITY;
```

In a more conventional approach in which low-level primitives are available for locking within data structures, the same structure would ordinarily be implemented by considering each entity as a data object which includes, in addition to any necessary local state information, an interlock to prevent concurrent access to that entity by more than one action. The individual actions would be coded as procedures, for example:

```
-- INTERLOCK operations defined in Section 3 --

type ENTITY is access
  record
    ACCESS_LOCK : INTERLOCK := UNLOCKED;
    -- local state fields --
  end record;


procedure ACTION1 (ENT : in ENTITY)
begin
  LOCK (ENT.ACCESS_LOCK);
  -- body of action 1 --
  UNLOCK (ENT.ACCESS_LOCK);
end ACTION1;

-- ACTION2 through ACTION10 are similarly defined --
```

The flavor of the two models above is very similar, particularly from the external point of view. In order to perform ACTION3 on some entity k in the task-based Ada approach, one issues the call

```
ACTION3 (k);
```

while in the interlock model, one performs

```
ACTION3 (pointer to entity k);
```

The semantic properties are also similar in the sense that each call is protected against the concurrent execution of other actions for that entity even though independent entities may be acted upon in parallel.

In the implementation of the two mechanisms above, however, there is a considerable disparity in the storage requirements for the control information which arises from the fact that the interlock model views the entities (data) and the actions (procedures) as entirely distinct units. In the task model, each entity in the task family has, as part of its structure, each of the associated entries, which has a multiplicative effect on per entry storage requirements. For example, in the interlock model, there are 100 data locks used to manage concurrency; in the task model, this function is managed by 1000 (i.e., 100 x 10) entries. Since each entry must include at least a queue pointer, this approach is clearly inefficient in terms of storage.

It is possible to design the task structure for a particular application in such a way that this cost is eliminated. For example, in the code sequence below there are only 100 entries to perform the necessary actions.

```
type ACTION is (ACTION1, ACTION2, ..., ACTION10);

task ENTITY (1 .. 100) is
  entry PERFORM_ACTION (ACT : in ACTION);
end ENTITY;
```

```
task body ENTITY is
begin
  loop
    accept PERFORM_ACTION (ACT : in ACTION) do
      case ACT of
        when ACTION1 =>
          begin
            -- body of action 1 --
          end;
        when ACTION2 =>
          begin
            -- body of action 2 --
          end;
        -- when clauses for remaining actions --
        when ACTION10 =>
          begin
            -- body of action 10 --
          end;
      end case;
    end PERFORM_ACTION;
  end loop;
end ENTITY;
```

While the above solution has the desired effect of reducing the storage requirements, it seems clear that the overall structure has been sacrificed and that the resultant program is considerably less natural than the earlier form in which each entry within a given task could be referred to in a procedural sense. It may be possible for the compiler to perform some optimization of this kind from the source code specification, but this seems like an exceptionally complex problem.

6.2.3 Implementation of Interlocks in Ada

Although we believe that the rendezvous mechanism described for Ada is quite powerful and provides the same functionality as programmer-accessible interlocks within the data structure, we feel that such interlocks will prove necessary in order to allow multiprocessor systems to be implemented with the required level of efficiency. The two preceding sections demonstrate that the interlock model is considerably more efficient than a straightforward implementation of the rendezvous scheme, in terms of both run-time efficiency and memory utilization. Because we feel that efficiency is of critical importance in most multiprocessor environments, we are concerned that the failure of Ada to provide adequate facilities for low-level interlocks will considerably reduce the overall applicability of the language.

We also believe that low-level facilities for managing interlocks can be added to the language without any significant change in the underlying structure of Ada. One possibility is simply to incorporate the data type INTERLOCK and the procedures LOCK and UNLOCK (as defined in Section 3.2.2) directly into the Ada language. This solution is certainly sufficiently general to satisfy the efficiency considerations and does so with a very minimal impact on the Ada language. A second alternative would be to define a new statement form, such as the <u>region</u> statement from Brinch Hansen [BrinchHansen73], which has the effect of insuring mutual exclusion on a particular interlock throughout a sequence of statements. This alternative offers greater protection against improper use of interlocks at the cost of introducing new syntactic forms into the Ada language.

At this point, it is important to note that the implementation of semaphore operations through the use of a generic task (as suggested in the Ada Reference Manual) is not a sufficient solution to the mutual exclusion problem, even if these primitives are implemented using special hardware support. There are two problems associated with the P and V operations as defined in Ada. First, tasks (including these generic tasks) are not part of the data environment. One of the principal uses of an interlock in conventional systems is to protect some structure from concurrent access. In Ada, there is no convenient way to associate a semaphore with a specific data object. The object may not explicitly contain a semaphore nor may it point to it in the access type sense. The best achievable solution is to use integer indices within the object to select the appropriate member of a semaphore family in a relatively cumbersome and obscure way.

The second problem stems from the FIFO semantics of the rendezvous mechanism in Ada. Although the Ada Reference Manual (page 9-11) notes that the fact that semaphores are "predefined authorizes an implementation to recognize them and implement them making optimal use of the facilities provided by the machine or the underlying system," presumably it is intended that the optimized forms of the P and V operations retain the semantics implied by their Ada definition. If this is true, semaphores are also required to obey a FIFO discipline. While this is not in any sense impossible, it does complicate the internal definition of the semaphore operations and requires the introduction of a queue for each semaphore.

6.3 Entries and the Name Problem

Another major problem in Ada stems from the manner in which processes are named. In Ada, tasks which perform some particular set of operations for separate internal data structures or devices are grouped together to form array-structured task families. In order to refer to a specific incarnation of a task, we must specify both the name of the task and the index of the specific process. Furthermore, since tasks in Ada are not data objects, we must supply the name field explicitly in the source code. This treatment of processes has several deficiencies when compared to other structures which allow a more flexible naming scheme.

6.3.1 Limitations of Array Functionality

One concern that arises from the naming convention for task families is that the array structure imposes a relatively arbitrary task structure which may not fit the nature of the particular application. Array structured task families are appropriate only when the process structure which they represent has a topology which behaves like an array. Other structures (particularly those which involve linked lists or other pointer-based structures) are cumbersome to implement in terms of a pre-supplied array structure. This problem is very similar to the problem of defining linked structures in Fortran or a similar language in which arrays are the primary compound structure.

As an example, let us again consider the case of the terminal concentrator example presented in Section 3.1.2. In this application, there is some large number of terminals of which only some relatively small fraction is likely to be connected at any given time. The activity for each terminal is monitored by a member of a task family which is assigned to that terminal as long as it is connected to the system. We will also assume that the total number of terminal tasks is constant (which allows them to be statically allocated) and that the association of terminals and tasks will change over time as terminals are connected and disconnected from the system. Ordinarily, there will be more terminal tasks than connected terminals at any particular time; these tasks remain idle until they are associated with a newly connected terminal.

In a structure such as this, the natural structure in which to store the idle terminal tasks is a linked free list. When a terminal is connected to the system, it is assigned to the first free task which is currently at the head of the list. When a terminal is disconnected, its associated process becomes idle and is linked onto the free list structure. These operations are extremely natural in a structure which permits pointer operations; when faced with an array structure, one is faced with the choice of (1) searching for free entries, (2) dynamically compactifying the task table so that the active tasks are contiguous, or (3) simulating the free list mechanism through the use of auxiliary arrays. These alternatives represent possible implementation strategies, but it is our contention that Ada prevents the most natural solution.

6.3.2 The Return Address Problem

A potentially more serious problem posed by the process naming convention is the "return address problem" which is briefly considered in the Ada Rationale (page 11-40). The concern here is that a server task has no way to reply to the calling task which requests service unless the identity of the calling task is known at translation time. The problem here is not one of authenticating a particular caller but rather one of identifying the calling task in some subsequent entry call.

Consider the case of a task whose function is to encrypt a message supplied by a caller and to return the encrypted message. In Ada, the canonical task description for this type of server is illustrated below:

```
task ENCRYPTION_SERVER is
  PACKET_SIZE : constant INTEGER := 256;
  type PACKET is array (1 .. PACKET_SIZE) of CHARACTER;
  entry SEND_NORMAL_MESSAGE (MSG : in PACKET);
  entry GET_ENCRYPTED_MESSAGE (MSG : out PACKET);
end ENCRYPTION_SERVER;


task body ENCRYPTION_SERVER is
  BUF : PACKET;
begin
  loop
    accept SEND_NORMAL_MESSAGE (MSG : in PACKET) do
      BUF := MSG;
    end SEND_NORMAL_MESSAGE;

    -- code to encrypt data in BUF --

    accept GET_ENCRYPTED_MESSAGE (MSG : out PACKET) do
      MSG := BUF;
    end GET_ENCRYPTED_MESSAGE;
  end loop;
end ENCRYPTION_SERVER;
```

While the code above performs the encryption function in a straightforward way and allows arbitrary tasks to call the two entries, it may not be appropriate in all cases. One potential problem arises in entry definitions which make use of a select statement to allow the server task to wait for a number of possible events. Because the select statement can appear only within the body of the called task, there is an inherent asymmetry in the tasking structure. Suppose that the programmer using ENCRYPTION_SERVER wanted a task with the following logical structure:

```
task body CALLING_TASK is

  -- code which generates PLAINTEXT for encryption --

  SEND_NORMAL_MESSAGE (PLAINTEXT);
  loop
    exit when ENCRYPTION_DONE;
    -- do some other work --
  end loop;
  GET_ENCRYPTED_MESSAGE (CODED_MESSAGE);

  -- code to make use of CODED_MESSAGE --

end CALLING_TASK;
```

While it is not possible to code the calling task in this way directly (because there is no way to transmit the ENCRYPTION_DONE signal in this fashion), this type of operation can be achieved if the roles of entry call and accept statement are reversed for the *GET_ENCRYPTED_MESSAGE entry as* in the recoded example below:

```
task ENCRYPTION_SERVER is
  PACKET_SIZE : constant INTEGER := 256;
  type PACKET is array (1 .. PACKET_SIZE) of CHARACTER;
  entry SEND_NORMAL_MESSAGE (MSG : in PACKET);
end ENCRYPTION_SERVER;
```

```
task body ENCRYPTION_SERVER is
  BUF : PACKET;
begin
  loop
    accept SEND_NORMAL_MESSAGE (MSG : in PACKET) do
      BUF := MSG;
    end SEND_NORMAL_MESSAGE;

    -- code to encrypt data in BUF --

    GOT_ENCRYPTED_MESSAGE (BUF);
  end loop;
end ENCRYPTION_SERVER;


task body CALLING_TASK is

  -- code which generates PLAINTEXT for encryption --

  SEND_NORMAL_MESSAGE (PLAINTEXT);
  loop
    select
      accept GOT_ENCRYPTED_MESSAGE (MSG : in PACKET) do
        CODED_MESSAGE := MSG;
      end GOT_ENCRYPTED_MESSAGE;
      else
        -- do some other work --
    end select;
  end loop;

  -- code to make use of CODED_MESSAGE --

end CALLING_TASK;
```

Unfortunately, this organization is only effective if there is a single calling task or at most a single family of callers. In the case that the calling task is a member of a task family, the caller can pass the index of the particular member as an additional argument to SEND_NORMAL_MESSAGE and then use this index in the subsequent GOT_ENCRYPTED_MESSAGE call, as in

```
CALLING_TASK(TASK_INDEX)'GOT_ENCRYPTED_MESSAGE (BUF);
```

It is impossible to write ENCRYPTION_SERVER as a general utility package which is available for use with any task that calls SEND_NORMAL_MESSAGE and defines an entry GOT_ENCRYPTED_MESSAGE for the reply. Because it is impossible to pass the identity of the calling task to ENCRYPTION_SERVER, there is no way for the server task to return the message to the appropriate caller since GOT_ENCRYPTED_MESSAGE is no longer uniquely defined. This is an unfortunate restriction since it seems to preclude the development of task libraries comparable to subroutine libraries in a well-organized environment for software development.

6.3.3 Tasks as Data Objects

The obvious solution to both the array topology problem and the return address problem is to consider individual activations of tasks to be data objects which can be incorporated into arbitrary structures or passed as parameters to server tasks. This issue is briefly discussed in the Ada Rationale (page 11-39) and the notion of anonymous activation variables from the language Tartan is introduced. Such a mechanism could be incorporated into Ada if it were possible to overcome the additional problems associated with the notion of task variables. For example, assume that all activations of tasks are data objects in the space of the type ACTIVATION_NAME and that each task implicitly defines the variable MY_NAME to be an identification of that activation.

The discussion of activation variables in the Ada Rationale correctly observes that the introduction of untyped task variables raises questions of strong typing similar to those found with procedure parameters in languages such as ALGOL-60. For example, even though the task definition

```
task body GENERAL_SERVER is
  DATA           : PACKET;
  RETURN_ADDRESS : ACTIVATION_NAME;
begin
  accept SERVER_REQUEST (T : in ACTIVATION_NAME,
                         INPUT : in PACKET) do
    DATA := INPUT;
    RETURN_ADDRESS := T;
  end SERVER_REQUEST;

  -- perform appropriate manipulation on DATA --

  RETURN_ADDRESS'REPLY(DATA);
end GENERAL_SERVER;
```

solves the return address problem, the use of an untyped process variable T is dangerous in the sense that we have no way to guarantee that the process referred to by T has a REPLY entry or that its parameter structure is compatible.

This problem, however, may be solved by eliminating the untyped activation variables in favor of a strongly typed system of specific entry variables. For example, assume that the reserved word entry is also usable as a type generating function in a similar fashion as array. It is then possible to declare a return address with no type ambiguity as illustrated below:

```
task body GENERAL_SERVER is
  DATA           : PACKET;
  RETURN_ADDRESS : entry (in PACKET);
begin
  accept SERVER_REQUEST (T : in entry (in PACKET),
                         INPUT : in PACKET) do
    DATA := INPUT;
    RETURN_ADDRESS := T;
  end SERVER_REQUEST;

  -- perform appropriate manipulation on DATA --

  RETURN_ADDRESS(DATA);
end GENERAL_SERVER;
```

In this case, the caller would issue the entry call

```
SERVER_REQUEST (MY_NAME'REPLY, INPUT_DATA);
```

thereby giving the complete (and unambiguous) address of the return entry.

There are other possible approaches to this problem; we have suggested the above scheme of strongly typed entry parameters not as an optimal solution but in order to demonstrate that strong typing considerations alone are not a sufficient justification for disallowing references to process activations within the data structure. We believe that the ability to code a general server with the ability to correctly address a reply is of major importance to the design of a rationally structured parallel control facility. Consequently, we believe that some mechanism for performing this function should be determined and incorporated into the Ada language.

6.4 Flexibility in the Scheduling Discipline

One additional area of concern that has developed during our study of Ada is the question of whether the scheduling discipline provided by the language is sufficiently general to support applications with important timing constraints. In particular, we are concerned that Ada does not provide adequate control over the scheduling strategy and that the scheduling algorithm is likely to encounter a number of problems associated with "cooperative scheduling."

To illustrate this problem, imagine that Ada is chosen as the implementation language for the design and development of a timesharing system for a multiprocessor. It is convenient in such a system to represent the individual user processes as independent tasks in the timesharing structure. In order to achieve fairness, timesharing systems typically limit the run-time allowed to a process to some maximum unit of time. If this time period (or quantum) is exceeded, the process is forcibly descheduled to allow other processes to run. The performance of the typical timesharing system is quite sensitive to the size and dynamic behavior of this quantum limit and it is important to be able to adjust this mechanism to conform to the loading demands.

In Ada, there is no apparent way to specify a run-time limit for a task nor is it possible for one task to control the scheduling or descheduling of another. Without this flexibility, it appears that there are only two possible schemes to provide fairness in a timesharing scheduler:

(1) Depend on the Ada scheduling discipline for all scheduling and descheduling operations and insure that the built-in mechanism provides all of the desired flexibility, presumably expressed in the form of pragma declarations to the compiler.

(2) Design a scheduler which operates "cooperatively" in the sense that the tasks themselves participate in the scheduling decisions. In this case, each task would be required to periodically check its accumulated run-time and dismiss itself through the use of a delay statement.

Obviously, each of the approaches outlined above is totally unacceptable for a timesharing application. The first either requires the system designer to change the structure of the implementation language or forces the system to make use of a built-in scheduling discipline which may be hopelessly inadequate to perform the more complex scheduling operations required of a timesharing system.

The second approach is equally unworkable in that it requires the compiler to perform complex path analysis and assemble code to poll the scheduler at acceptably frequent intervals. This problem is similar to the one raised by the existence of strips in the BBN Pluribus (see Section 2.4.3). The problems that arise in this type of scheduling are so severe that this alternative tends to be rejected out of hand. In his assessment of the process scheduling facility in Ada [Lamb79], Paul Hilfinger writes:

> It seems that the tasks being scheduled must be written to be aware of the fact that they are being scheduled, and to do appropriate sends or procedure calls at intervals. This is a violation of abstraction; no reasonable operating system in existence requires that its processes cooperate to be scheduled.

There are several potential approaches to this problem which affect the structure of the language to varying degrees. Perhaps the most straightforward mechanism is to allow one task to forcibly deschedule another task. This would provide a monitoring task with at least some primitive ability to control the scheduling discipline. This could be implemented through the addition of a new primitive such as

```
deschedule T;
```

or as an extension of the priority mechanism. If one task were allowed to alter the priority of another and changes in priority were implemented so as to force a scheduler transition, one might begin to have an acceptable facility for scheduling control.

Again, the above scheme is not intended to be either complete or in any sense optimal. In this discussion, our concern has been to identify a problem area and propose a potential direction for solution.

7. CONCLUSIONS

In this report, we have argued that multiprocessor systems are frequently used for real-time applications in which run-time efficiency requirements are of critical importance. For this reason, we believe that the design of a high-level language system which is intended for use in real-time, multiprocessor-based applications must be sensitive to these requirements and must allow the programmer to write code which satisfies the efficiency constraints imposed by the application.

The need to produce highly efficient code is well understood by those who have experience in designing real-time applications and is reflected in the technical requirements for a common high order language for the Department of Defense. Section 1D of the Steelman requirements [DOD78] specifies that language "features should be chosen to have a simple and efficient representation in many object machines." Moreover, Steelman recognizes that the tasking facility is particularly subject to such efficiency considerations in its requirement (Section 9B) that the "parallel processing facility shall be designed to minimize execution time and space."

We believe that the Ada language, as currently designed, does not meet these needs for several reasons:

(1) The use of a complete rendezvous system results in unnecessary scheduling delays. This problem is particularly severe in the relatively important case of message passing in that Ada requires the sender of a message to wait for the scheduler before it is allowed to proceed. This structure is considerably less efficient than message passing systems implemented with queues and imposes a relatively high cost on the use of a particularly important communication discipline.

(2) Ada does not provide sufficient flexibility in its process control structure to allow the programmer to choose the mechanism most closely suited to the requirements of the application. In particular, the fact that the mutual exclusion mechanism is associated with the control structure rather than the data structure leads to convoluted program structures or serious inefficiencies in the use of space.

(3) The naming conventions used to indicate specific processes in Ada are not sufficiently general to allow the programmer to represent process structures which accurately reflect the underlying structure of the algorithm. Moreover, the fact that no general mechanism exists to allow one process to communicate its identity to other processes in the system severely limits the modularity of the task structure.

(4) The language does not provide the user with sufficient control over the scheduling discipline.

Each of the criticisms listed above is presented in detail in Section 6 of this report. For each of these deficiencies, alternative structures are proposed which would allow Ada to satisfy these objections without requiring extensive redesign of the language as a whole.

In summary, we wish to emphasize that the parallel processing facilities currently provided by Ada do not satisfy the requirements of real-time systems such as those typically chosen for implementation on a multiprocessor. On the other hand, we feel that good solutions do exist for most of the problems that we have identified and that those solutions can be incorporated into Ada with relatively little change to the overall structure of the code. Based on our experience with multiprocessors and real-time systems, we feel that the efficiency cost implied by the current Ada design severely limits the extent to which Ada is acceptable for real-time applications. We strongly urge that modifications such as those suggested in this report be incorporated into Ada to increase its utility in this important area of application.

REFERENCES

[Bartlett77] Bartlett, Joel F., _A "NonStop" Operating System_, Tandem Computers Inc., Cupertino, California, 1977.

[BBN75] Bolt Beranek and Newman Inc., _Pluribus Documents_, Volumes 1-5, 1975.

[BrinchHansen73] Brinch Hansen, Per, _Operating System Principles_, Prentice Hall, Englewood Cliffs, New Jersey, 1973.

[Clarke79] Clarke, Edmund M., "Approximate Algorithms for the Optimizing of Busy Waiting in Parallel Programs," Proceedings of the Twentieth Annual IEEE Symposium on the Foundations of Computer Science, 1979.

[Dijkstra68] Dijkstra, E. W., "Co-operating Sequential Processes," published in Genuys, F., _Programming Languages_, Academic Press, London, 1968.

[DOD78] Department of Defense, "STEELMAN -- Requirements for High Order Computer Programming Languages," Defense Advanced Research Projects Agency, Arlington, Virginia, June 1978.

[Evans77] Evans, Arthur Jr. and C. Robert Morgan, _Communications Oriented Language (COL): Language Definition_, BBN Report 3534, May 1977.

[Fuller78] Fuller, Samuel H. and Samuel P. Harbison, _The C.mmp Multiprocessor_, Computer Science Department Report CMU-CS-78-146, Carnegie-Mellon University, October 1978.

[Heart73] Heart, Frank E., Severo M. Ornstein, William R. Crowther and William B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," _AFIPS Conference Proceedings_, Volume 42, June 1973, pp. 529-537.

[Hoare72] Hoare, C.A.R., "Towards a Theory of Parallel Programming", published in Hoare, C.A.R. and R.H. Perot, editors, _Operating Systems Techniques_, Academic Press, London, 1972.

[Hoare73] Hoare, C.A.R., "Monitors: an Operating Systems Structuring Concept", IFIP-WG 2.3, Munich, 1973.

[Hoare77] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, Volume 21, Number 8, August 1978.

[Ichbiah79] Ichbiah, Jean D., Jean-Claude Heliard, Olivier Roubine, John G.P. Barnes, Bernd Krieg-Brueckner and Brian A. Wichmann, Rationale for the Design of the Ada Programming Language, published in the ACM SIGPLAN Notices, Volume 14, Number 6, June 1979.

[Jensen75] Jensen, Kathleen and Niklaus Wirth, PASCAL User Manual and Report, Second Edition, Springer-Verlag, New York, 1975.

[Jones78] Jones, Anita K., Robert J. Chansler, Jr., Ivor Durham, Peter H. Feiler, Donald A. Scelza, Karsten Schwans and Steven R. Vegdahl, "Programming Issues Raised by a Multiprocessor," Proceedings of the IEEE, Volume 66, Number 2, February 1978.

[Katsuki78] Katsuki, David, Eric S. Elsam, William F. Mann, Eric S. Roberts, John G. Robinson, F. Stanley Skowronski and Eric W. Wolf, "Pluribus -- An Operational Fault-Tolerant Multiprocessor," Proceedings of the IEEE, Volume 66, Number 10, October 1978.

[Katzman77] Katzman, James A., A Fault-Tolerant Computing System, Tandem Computers Inc., Cupertino, California, 1977.

[Knuth74] Knuth, Donald E., "Structured Programming with GOTO Statements," Computing Surveys, Volume 6, Number 4, December 1974.

[Lamb79] Lamb, David A., editor, "Commentary on the RED and GREEN Candidates for the Ada Language," Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April, 1979.

[Nestor79] Nestor, John and Mary Van Deusen, RED Language Reference Manual, Intermetrics, Cambridge, Massachussets, 1979.

[Newell75] Newell, Alan and George Robertson, Some Issues In Programming Multi-Mini-Processors, Department of Computer Science, Carnegie-Mellon University, January 1975.

[Oleinick78] Oleinick, Peter N. and Samuel H. Fuller, The Implementation and Evaluation of a Parallel Algorithm on C.mmp, Computer Science Department Report CMU-CS-78-125, Carnegie-Mellon University, June 1978.

[Ornstein75] Ornstein, Severo M., William R. Crowther, Michael F. Kraley, Robert D. Bressler, Anthony Michel, and Frank E. Heart, "Pluribus -- A Reliable Multiprocessor," AFIPS Conference Proceedings, Volume 44, May 1975, pp. 551-559.

[Robinson78] Robinson, John G., and Eric S. Roberts, "Software Fault-Tolerance in the Pluribus," Proceedings of the 1978 National Computer Conference, June 1978, pp. 563-569.

[Schmid76] Schmid, H.A., "On the Efficient Implementation of Conditional Critical Regions", Acta Informatica, Number 6, 1976.

[Siewiorek78] Siewiorek, Daniel P., Vittal Kini, Henry Mashburn, Stephen McConnel and Michael Tsao, "A Case Study of C.mmp, Cm*, and C.vmp: Part 1 -- Experiences with Fault Tolerance in Multiprocessor Systems," Proceedings of the IEEE, Volume 68, Number 10, October 1978, pp. 1178-1199.

[Swan77a] Swan, Richard J., Samuel H. Fuller and Daniel P. Siewiorek, "Cm* -- A Modular, Multi-Microprocessor," Proceedings of the National Computer Conference, June 1977.

[Swan77b] Swan, Richard J., Andy Bechtolsheim, Kwok-Woon Lai and John K. Ousterhout, "The Implementation of the Cm* Multi-Microprocessor," Proceedings of the National Computer Conference, June 1977.

[Williams72] Williams, R. K., "System 250 -- Basic Concepts," in Proceedings of the I.E.R.E Conference on Computers -- Systems and Technology, London, England, 1972.

[Wolverton74]	Wolverton, R. W., "The Cost of Developing Large-Scale Software," <u>IEEE Transactions on Computers</u>, June 1974.
[Wulf72]	Wulf, William A. and C. G. Bell, "C.mmp -- A Multi-Mini-Processor," <u>1972 AFIPS Conference Proceedings</u>, Volume 41, pp. 765-777.
[Wulf73]	Wulf, William A., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack, <u>HYDRA: The Kernel of a Multiprocessor Operating System</u>, Computer Science Department Report, Carnegie-Mellon University, June 1973.
[Wulf78]	Wulf, William A. and Samuel P. Harbison, <u>Reflections in a Pool of Processors</u>, Computer Science Department Report CMU-CS-78-103, Carnegie-Mellon University, February 1978.

LMED

8-81